Universidad Autónoma de Madrid
Escuela politécnica superior

Double Degree in Mathematics and Computer Engineering

Final Degree Project

HARDWARE PARALLELIZATION OF CORES ACCESSING MEMORY WITH IRREGULAR ACCESS PATTERNS

Author: Carlos Alfaro
Tutor: Fabrizio Ferrandi
Supervisor: Javier Garrido Salas

February 2018

HARDWARE PARALLELIZATION OF CORES ACCESSING MEMORY WITH IRREGULAR ACCESS PATTERNS

Author: Carlos Alfaro
Tutor: Fabrizio Ferrandi
Supervisor: Javier Garrido Salas

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano (Italy)
February 2018

Abstract

Abstract

This project studies FPGA-based heterogeneous computing architectures with the objective of discovering their ability to optimize the performances of algorithms characterized by irregular memory access patterns. The example used to achieve this is a graph algorithm known as Triad Census Algorithm, whose implementation has been developed and tested.

First of all, the triad census algorithm is presented, explaining the possible variants and reviewing the existing implementations upon different architectures. The analysis focuses on the parallelization techniques which have allowed to boost performance, thus reducing execution time. Besides, the study tackles the OpenCL programming model, the standard used to develop the final application. Special attention is paid to the language details that have motivated some of the most important design decisions.

The dissertation continues with the description of the project implementation, including the application objectives, the system design, and the different variants developed to enhance algorithm performance.

Finally, some of the experimental results are presented and discussed. All implemented versions are evaluated and compared to decide which is the best in terms of scalability and execution time.

Key words

Algorithms, parallel programming, heterogeneous architectures, OpenCL, FPGA, computational complexity, social networks, graph mining.

Resumen

Este proyecto estudia las arquitecturas de ordenadores heterogéneas basadas en FPGA con el objetivo de descubrir su capacidad para optimizar las prestaciones de algoritmos caracterizados por accesos irregulares a memoria. Para ello, se utiliza como ejemplo un algoritmo de grafos, conocido como algoritmo de censo de las tríadas, del cual se ha desarrollado la implementación y se han evaluado las prestaciones.

En primer lugar, se presenta teóricamente el algoritmo de censo de las tríadas, explicando los diferentes versiones existentes y analizando las implementaciones del mismo sobre diversas arquitecturas. El estudio se centra en las técnicas de paralelización que han permitido aumentar la eficiencia del algoritmo y reducir así su tiempo de ejecución. Además, se resumen los conceptos básicos de OpenCL, el estándar de programación que ha sido utilizado para desarrollar la aplicación final. Se presta especial atención a los detalles del lenguaje que han resultado determinantes en las decisiones de diseño tomadas.

La disertación continúa con la descripción de la implementación del proyecto: se explican los objetivos de la aplicación, el diseño de la misma y las múltiples variantes que se han desarrollado para tratar de mejorar la eficiencia del algoritmo.

Finalmente se discuten los resultados obtenidos. Se evalúan todas las versiones implementadas, y se comparan las prestaciones de cada una de ellas para decidir cuál es la mejor en cuanto a escalabilidad y tiempo de ejecución.

Palabras Clave

Algorítmica, programación paralela, aplicaciones heterogéneas, OpenCL, FPGA, complejidad computacional, redes sociales, minería de grafos.

Acknowledgements

First and foremost, I would like to thank my parents, who have always supported and helped me throughout my educational journey.

I would also like to mention my teachers from my school *John Henry Newman* and university, *Universidad Autónoma de Madrid*. They have shown me the beauty of reality and the passion of discovering its nature.

Last but not least, I wish to express my sincere thanks to my tutor, professor Fabrizio Ferrandi, and Ph.D. Marco Lattuada, who have allowed me to collaborate with them to develop this work.

Deo Gratias.

Contents

Figure Index

Table Index

1 Introduction

1.1 Triadic Analysis as a local-pattern graph mining method

Used to represent relationships and behaviors, graphs are powerful mathematical objects that have a broad range of applications in many different domains. In particular, directed graphs are commonly used to represent different kinds of networks such as social relationships, paper citations, and the World Wide Web, among others.

Multiple research studies have demonstrated that the characteristics of real-world networks differ considerably from a uniformly distributed graph, and present global and local patterns whose study reveals valuable information about the nature of the relationships present in it.

In global terms, it has been shown that these networks possess a particular structure, which is commonly known as *scale-free* structure [1]. The essential characteristic of a scale-free network is that its degree distribution is not uniform, but follows a power law[1]. In other words, in these networks there are few nodes with a high degree, while the vast majority of them have a very low degree.

In local terms, many networks present patterns that help to understand the nature of the relationships among the entities under study, for example symmetry, reciprocity or transitivity. One of the most important local pattern identification technique is **triadic analysis**. Triadic analysis encompasses a set of graph mining methods that allow studying the characteristics of the interactions between each threesome of nodes within the graph [2]. Triadic analysis has revealed to yield significant results in the fields of sociology, economics or security among others [3] [4] [5] [6].

In particular, the most important method is known as **triad census**. This method, as explained later in this report, examines the distribution of all the different relationships possible between each triple of actors present in the network.

A real social network can grow to millions of nodes and edges, making the computation of the triad census a very challenging task in terms of execution time and resource consumption. Therefore, its implementation in high-performance computer architectures could facilitate the labor of researchers and network analysts.

[1] $p(x) = ax^{-k}$, where a and k are constants

1.2 Graph Analysis Challenges

Implementing graph algorithms and achieving good performance represents a great challenge due to their memory-bound nature. Given the intrinsic structure of the graphs, it is possible that connected nodes are placed at very distant positions in memory. In graph processing applications, thus, memory accesses follow irregular patterns that cannot be predicted at programming or compile time. As a result, these applications cannot exploit spatial locality and memory hierarchy (e.g. caches) to enhance performance. Real-world networks tend to be very large and complex, making memory accesses frequent and time-consuming.

Furthermore, the gap between memory and processor speeds are increasing; i.e. the time needed to copy data from memory to the registers is much more than the time the CPU spends processing those data. This involves that most of the execution time of a program is spent waiting for data rather than effectively performing computations, which brings about a poor CPU usage. In order to overcome memory latency issues and optimize processor usage, the simplest solution is to spawn multiple tasks that share the processing resources. Thus, even if one process has to stall and wait for an I/O operation, the other processes can continue using the processing units.

Working with real scale-free networks, though, enforces to be especially cautious when distributing the workload among the tasks launched. Given the enormous degree differences, a straightforward policy of assigning the same number of nodes to be processed by each task might not enhance performance significantly.

1.3 Heterogeneous computing using FPGA accelerators

In the last decades of the XX century, the single-processor computing power increased constantly and exponentially, following the well-known pattern given by Moore's Law [7]. In the last years, though, the capacity of single-core processors has encountered its limits due to three main reasons:

1. **The Power Wall:** The physics of the hardware impede to increment the number of transistors per area unit while maintaining a reasonable power consumption. As clock frequency increases, heating soars to unacceptable levels, risking to burn the device.

2. **Instruction-level Parallelism Wall:** It is very difficult to extract parallelism from a single instruction stream by relying on the compiler and scheduler.

3. **The Memory Wall:** As mentioned in the former section, there is an increasing gap between processor and memory speeds. This fact limits the performance of the system and makes useless the efficiency of the processors. Memory hierarchy and caches can only help to a certain extent.

Nowadays, performance needs to be extracted from **parallelism**. This is the reason why all modern CPUs contain multiple cores, and the most advanced systems are composed of a lot of high-speed processors. Anyway, to obtain the best performance it is not sufficient to send tasks to execution in parallel. It is also necessary to assign each task to the most convenient execution unit.

Software applications perform different tasks which could be divided into three main categories:

- **Control-intensive tasks:** They include many instructions to control the execution flow. E.g. searching, parsing, etc. These tasks run best on superscalar CPUs.
- **Data-intensive tasks:** They require the process of great amounts of data. E.g. Image processing, data mining, etc. These tasks perform well on vector architectures such as GPUs
- **Performance-intensive tasks:** They contain heavy computations and calculations. E.g. Iterative methods, financial modeling, etc. These tasks run well on hardware especially designed and optimized for such computations.

Therefore, in order to boost execution performance of such applications, it is necessary to deploy them over heterogeneous computing environments including different components such as CPUs, GPUs, and FPGAs [8].

Programming applications to run on a heterogeneous architecture is normally a very hard task, since developers need to write dedicated source code for each hardware component and be able to orchestrate the different elements to build a single software unit. In particular, to achieve fine-grained parallelism on FPGA devices, it is necessary to directly design the hardware by writing on a *Hardware Description Language* (HDL). Thus, working in such environments requires a broad range of programming skills and platform-specific knowledge. Besides, programming and debugging times tend to be long, increasing the product time to market. It is clear, therefore, the need for a cross-platform parallel programming standard model that increases the level of abstraction and hides platform-specific issues. Among the different standards developed, a very important one is OpenCL [9].

The purpose of the work developed[2] and presented in this report is to explore the behavior and performances of an FPGA-based heterogeneous architecture when running applications that exhibit irregular access patterns. In particular, the project implements different versions of the **triad census algorithm** and explores different techniques to optimize performance.

The programming environment used throughout the project has been the *Intel® FPGA SDK for OpenCL*, an implementation of the OpenCL standard that includes the OpenCL APIs and the necessary tools to synthesize the code that will be deployed on the FPGA.

The report begins presenting the triad census algorithm and analyzing the state of the art regarding its implementation. Chapter 3 summarizes the key concepts of the OpenCL standard, which may help to understand the project developed and some of the decisions taken. Chapter 4 describes the design an implementation of the project, and chapter 5 presents some of the experimental performance results obtained. The report concludes with some interesting topics about possible future developments.

[2]The project code is publicly available in the GitHub repository `https://github.com/carlosalfaro94/triad_census_on_FPGA.git`. It can be downloaded and used following the guidelines given in Appendix C.

2

State of the Art

This chapter analyzes the different implementations of the triad census algorithm that have served as the background to construct the final solution of this work.

The first part of the chapter is devoted to explain what is intended by triad census, and to define the basic concepts and definitions necessary for the rest of the chapter. The discussion continues with the sequential implementations of the triad census algorithm. Finally, some of the last work made on this algorithm is reviewed, focusing on the implementation techniques (including parallelization) that allow boosting performance.

2.1 Introduction: Preliminary definitions

2.1.1 Graph definitions

A **directed graph** (or **digraph**) is an ordered pair $G = (V, E)$, where V is a nonempty set of **vertices** or nodes, and $E \subset V \times V$ is the set of **directed edges**. Each directed edge is an **ordered** pair of vertices.

Let $G = (V, E)$ be a digraph, with $|V| = n$ and $|E| = m$. Let $u, v \in V$.

1. v is **adjacent** to u if $(u, v) \in E$
2. $Adj^{\rightarrow}(u) := \{v \in V \mid (u, v) \in E\}$; $deg^{\rightarrow}(u) := |Adj^{\rightarrow}(u)|$
3. $Adj^{\leftarrow}(u) := \{v \in V \mid (v, u) \in E\}$; $deg^{\leftarrow}(u) := |Adj^{\leftarrow}(u)|$
4. $Adj(u) := Adj^{\rightarrow}(u) \cup Adj^{\leftarrow}(u)$; $deg(u) := |Adj(u)|$
5. **Handshaking lemma**: The following property holds :

$$\sum_{u \in V} deg(u) = 2m$$

6. The **average degree** in graph G is $k = \frac{\sum_{u \in V} deg(u)}{\sum_{u \in V} u} = \frac{2m}{n}$.

7. The **degree distribution** of graph G is the probability distribution of the node degrees.
8. G is **connected** if there are no isolated nodes in G. The minimum number of edges necessary to have a connected graph is $m = n - 1$.
9. The maximum number of edges that a graph can have is $m = n(n-1)$, thus being **complete**.
10. The **density** of graph G is $d = \frac{m}{n(n-1)}$. It is a number between 0 and 1.
11. A graph is considered to be **sparse** when its number of edges is $m = O(n)(\Rightarrow k = O(1))$, and is considered to be **dense** when its number of edges is $m = O(n^2)(\Rightarrow k = O(n))$.

2.1.2 Triadic analysis

Given a digraph $G = (V, E)$, a **triad** is an unordered triple of vertices along with the edges that possibly connect them.
Given a triple of nodes, there are 6 possible edges that connect the triple. Therefore, there are $2^6 = 64$ possible triads. Since the triple is unordered (i.e the nodes are indistinguishable) the isomorphic cases can be removed, and only 16 different, non-isomorphic triads remain. Figure 2.1 lists all the possible non-isomorphic triad classes, tagging them in the following manner: The digits correspond to the number of null vertices, single vertices and double vertices respectively. In the cases when all these digits coincide, a letter (U, D, C, T) is furtherly added to indicate the nature of the triad (Up, Down, Circular and Transitive, respectively).

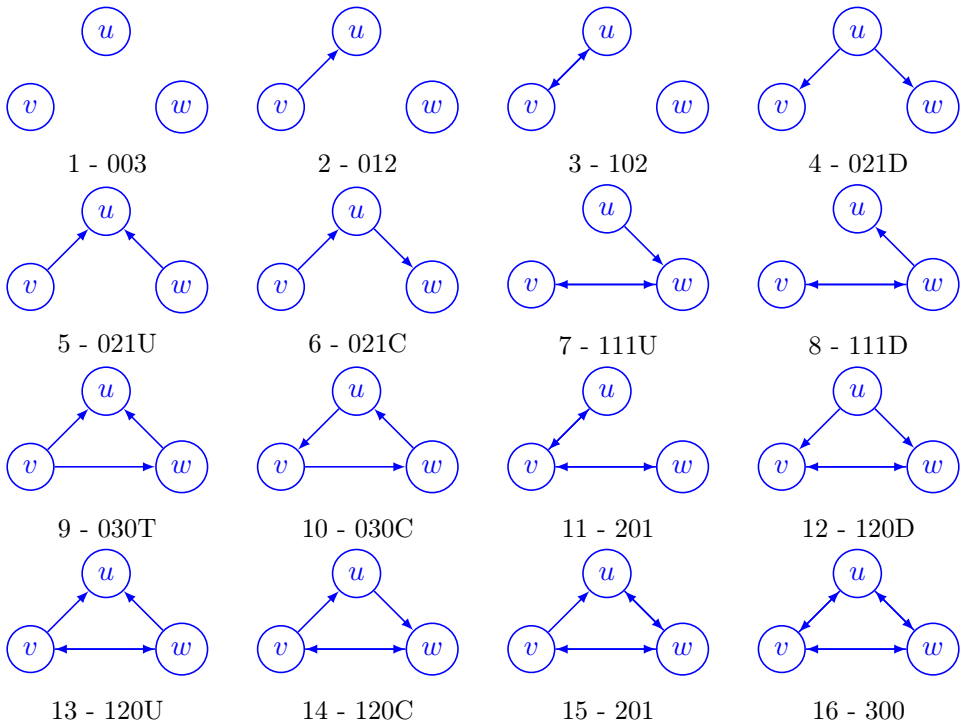

Figure 2.1: Classes of triads in a directed graph

The triad classes can be divided into three different groups:

1. *Null triads*: A set of 3 isolated nodes. The only one is class 1.
2. *Diadic triads*: A triad in which only two nodes are connected, and the other one is isolated. Classes within this group are 2 and 3.
3. *Connected triads*: A triad with no isolated nodes. The rest of the classes (4...16) fall into this group.

If $|V| = n$, there are $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ possible triads in G. The **triad census** of G is the count of each triad class on each possible triple of nodes in G.

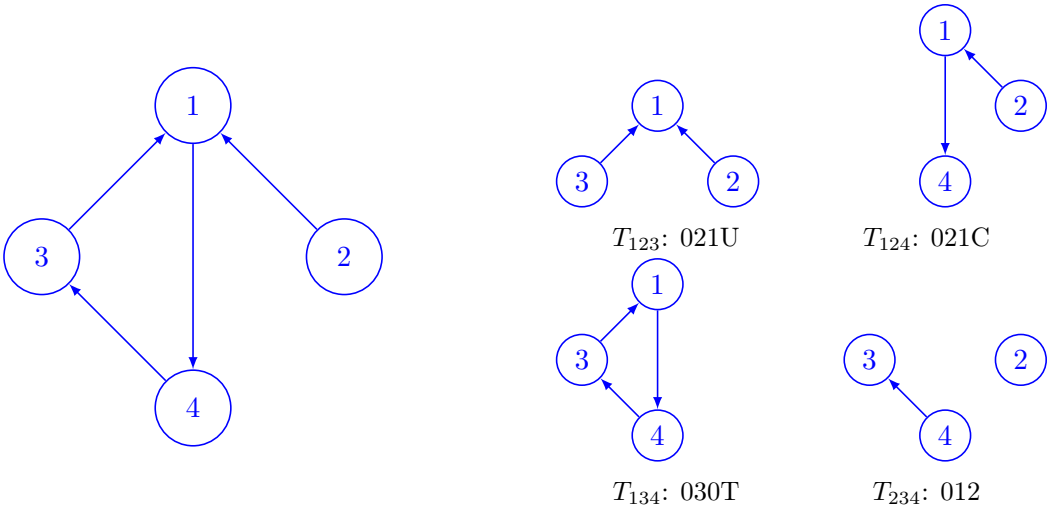

Figure 2.2: The four triads of a simple 4-node graph

2.2 Triad census algorithms

As explained before, the triad census algorithm computes the counts of each triad class on every triple of nodes of a given graph. In general, the algorithm will receive as input a graph G, and will produce a 16-position array containing the frequency of each triad type. There are two main version of this algorithm, the Brute Force approach and the Batagelj and Mrvar's approach.

2.2.1 Brute Force

The Brute Force (BF) triad census approach iterates over all possible triads of G to compute their triad type. In order to avoid considering the same triple (u, v, w) twice, nodes are tagged with a comparable id (for example, an unsigned integer), and only a canonical selection $u < v < w$ is considered. The Brute Force algorithm is shown in pseudocode 1.

As shown in the pseudocode, the algorithm makes use of another procedure, IsoTricode. This procedure returns the isomorphic code of the triad (u, v, w), which is the number between 1 and 16 that corresponds to the taxonomy showed in figure 2.1. This code is then used to index the *Census* array and increment the count by 1.

The Brute Force algorithm has a computational complexity if $O(n^3)$ for all undirected graphs.

Algorithm 1 Brute Force Triad Census

Input: $G = (V, E)$
Output: *Census* array with frequencies of triadic types

```
1: procedure TRIAD_CENSUS_BF(G = (V, E))
2:     for i from 1 to 16 do
3:         Census[i] ← 0                                   ▷ Initialize Census
4:     for u ∈ V do
5:         for v ∈ V if u < v do
6:             for w ∈ V if v < w do
7:                 TriType ← IsoTricode(u, v, w);
8:                 Census[Tritype] ← Census[Tritype] + 1;
9:     return Census
```

2.2.2 Subquadratic

The second algorithm version studied is the Batagelj and Mrvar's (BM) algorithm [10], which takes into account the adjacencies of the nodes to increment performance. In this case, the algorithm needs to receive the list of adjacencies of each node. The Batagelj and Mrvar's Subquadratic algorithm is presented in pseudocode 2:

Algorithm 2 Matagelj and Mrvar's Subquadratic Triad Census

Input: $G = (V, E)$; $N = \bigcup_{u \in V} Adj(u)$
Output: *Census* array with frequencies of triadic types

```
1: procedure TRIAD_CENSUS_BM(G = (V, E), N)
2:     for i from 1 to 16 do
3:         Census[i] ← 0                                   ▷ Initialize Census
4:     for u ∈ V do
5:         for v ∈ Adj(u) if u < v do
6:             S ← Adj(u) ∪ Adj(v) \ {u, v}
7:             if v ∈ Adj→(u) and u ∈ Adj→(v) then
8:                 tritype ← 3
9:             else
10:                tritype ← 2
11:            Census[tritype] ← Census[tritype] + n − |S| − 2
12:            for w ∈ S do
13:                if (v < w) or (u < w and w < v and w ∉ Adj(u)) then
14:                    TriType ← IsoTricode(u, v, w);
15:                    Census[Tritype] ← Census[Tritype] + 1;
16:    sum ← 0
17:    for i from 2 to 16 do
18:        sum ← sum + Census[i]
19:    Census[1] ← (1/6) n(n − 1)(n − 2) − sum
20:    return Census
```

The BM algorithm counts separately the *connected* triads, the *diadic* triads, and the *null* triads. As line 5 of the pseudocode shows, the algorithm considers connected diads (u, v) (respecting the canonical selection $u < v$). Then (line 6), computes and saves in S the union of the adjacency lists of u and v to consider *connected* triads $(u, v, \cdot)$ For the rest of nodes, which are not connected neither to u nor to v, the algorithm sums the respective count of *diadic* triads

(which is $n - |S| - 2$).

Note that, to count all *connected* triads only once, the algorithm respects the canonical selections $u < v < w$ and $\{u < w < v : w \notin Adj(u)\}$. (Because, if $u < w$ and $w \in Adj(u)$, it is considered in other iteration of the loop of line 5).

Finally, the *null* triads (position 1 of the *Census* array) are computed as the difference between the total number of triads and the sum of the counts of the rest of the triad types.

This approach is much more efficient than the Brute Force approach since computes triadic types based on the adjacency lists of the nodes. On sparse graphs, the BM algorithm performs in $O(m)$ time, which is a subquadratic performance.

Batagelj and Mrvar's algorithm is considered the state of the art (in terms of sequential performance) of the triad census algorithm.

2.3 Triadic census implementations

The aim of this section is to present a brief analysis of the previous work made over the implementation of the triad census algorithms, with special focus to the details that have been taken into account to develop the final solution of this work.

2.3.1 Implementing and evaluating Multithreaded Triad Census Algorithms on the Cray XMT

One of the first parallel implementations of the triad census algorithm was designed by Chin et. Al. [11]. Based on the Batagelj and Mrvar's algorithm, they develop a parallel implementation taking into account the balance of work among the different tasks launched.

In their implementation, they make use of the TASK_QUEUE_GENERATION procedure presented in pseudocode 3:

Algorithm 3 Task queue generation for parallel tasks triadic census algorithm

Input: $G = (V, E)$; $N = \bigcup_{u \in V} Adj(u)$
Output: D array of task queues

```
1:  procedure TASK_QUEUE_GENERATION(G = (V, E), N)
2:      i ← 1
3:      counter ← 0
4:      for u ∈ V do
5:          for v ∈ Adj(u) if u < v do
6:              D[i] ← D[i] ∪ (u, v)
7:              counter ← counter + |Adj(u)| + |Adj(v)|
8:              if counter > MaxNeighborSetSize then
9:                  i ← i + 1
10:                 counter ← 0
11:     return D
```

As the pseudocode illustrates, the procedure divides all the possible pairs $\{(u, v) : u < v\}$ present in G into different tasks, but controlling the number of possible connected triads that each task has to process (that cannot exceed a certain limit $MaxNeighborSetSize$)

The parallel implementation of the BM triad census algorithm is presented in pseudocode 4.

Algorithm 4 Parallel Tasks Subquadratic Triad Census

Input: $G = (V, E)$; $N = \bigcup_{u \in V} Adj(u)$; D array of task queues
Output: $Census$ array with frequencies of triadic types

```
1: procedure PARALLEL_TASKS_TRIAD_CENSUS(G = (V, E), N, D)
2:     for i from 1 to 16 do
3:         Census[i] ← 0                                   ▷ Initialize Census
4:     for T ∈ D do
5:         for (u, v) ∈ T do
6:             S ← Adj(u) ∪ Adj(v) \ {u, v}
7:             if v ∈ Adj→(u) and u ∈ Adj→(v) then
8:                 tritype ← 3
9:             else
10:                tritype ← 2
11:            Census[tritype] ← Census[tritype] + n − |S| − 2
12:            for w ∈ S do
13:                if (v < w) or (u < w and w < v and w ∉ Adj(u)) then
14:                    TriType ← IsoTricode(u, v, w);
15:                    Census[Tritype] ← Census[Tritype] + 1;
16:    sum ← 0
17:    for i from 2 to 16 do
18:        sum ← sum + Census[i]
19:    Census[1] ← (1/6) n(n − 1)(n − 2) − sum
20:    return Census
```

This algorithm was coded and optimized for running on a Cray XMT machine.

The **Cray XMT** is a multithreaded system specially designed to tolerate memory access latencies by switching context between threads. As a result, this machine has the potential of significantly improving the execution speed of irregular data-intensive applications, such as the one considered in the paper. Assuming all data and resource dependencies are met, a stream can be scheduled for instruction issue in a single cycle.

Evaluation and results

In order to compare the performance of each of the algorithms, the authors tested them using different graphs and resources.

The first tests were conducted upon a 16-processor XMT and compared the performances of the Brute Force algorithm, the BM algorithm, and the parallel-tasks BM algorithm. Preliminary tests using a random 10.000-node, 100.000-edges graph showed the great differences of execution time between the BF and the other BM approaches. The rest of the tests did not consider the BF approach.

Next, the authors explored different execution options using a much larger network consisting of 3.8 million nodes and 16.5 million edges. In the first case, they used *implicit parallelism*, thus relying on the compiler to automatically parallelize the loops in the code. In this case, the parallel tasks approach performed less efficiently than the sequential one, due to the overhead introduced by the sequential task queue generation. In the second case, they introduced XMT pragma constructs called *loop futures* to enforce explicit parallelism within the code. Thanks to these, the parallel tasks approach showed a significant performance improvement, becoming the fastest in terms of execution time and exhibiting the greatest speedup.

The authors continued to evaluate the performance and behavior of the algorithm in a larger XMT machine of 128 processors. For these tests, they created two random networks consisting respectively of 12 million nodes and 120 million edges; and 35 million nodes and 350 million edges. Again, the loop future parallel task approach exhibited the best time performance. In this case, though, the authors perceived a degradation in the speedup curve: when processor count exceeded 96, execution performance decayed. As the authors explain, this phenomenon is due to the network saturation. After some tests, the conclusion they came to was that execution is optimal when using around 9000 available hardware threads.

Finally, the authors discuss the CPU utilization profiles of both the implicit parallelism and loop future versions. In both of them, the triad census computation reaches around 50-55% of CPU usage. Anyway, the loop future version maintains this peak for a longer time, thus revealing a more efficient resource usage. In an attempt to improve the load balancing of the implicit parallelism version, the authors inserted the *interleave schedule* pragma in the code; a construct that allows assigning contiguous iterations to different compute units. This compiling option turns useful when executing triangular loops[1]. The interleaved schedule version showed to maintain a high CPU utilization and was comparable to the loop future version also in terms of execution time.

Brief Analysis

The main contribution of this paper to the parallelization of the triad census algorithm is the idea of subdividing the work into pairs of nodes (u, v) which are processed by different threads. In fact, the inner loop of the BM algorithm maintains fixed these two first nodes and iterates over their neighbors. The triad counts of the triples formed can be performed independently, using a SIMD parallelization technique.

Furthermore, the technique of using the $MaxNeighborSetSize$ variable is a simple but efficient solution to balance the load between the different threads.

2.3.2 Scalable Triadic Analysis of Large-Scale Graphs: Multi-Core vs Multi-Processor vs. Multi-Threaded Shared Memory Architectures

This paper is the continuation of the work made by Chin et al. [12] and described earlier. In this case, their purpose is to compare the parallel triad census algorithm in three different shared-memory systems:

1. **Cray XMT:** The machine used in the previous developments.
2. **HP Superdome:** This machine is a two cabinet, SD64 SX2000CEC with 8 cells per cabinet, each with 4 sockets, equipped with 1.6GHz dual-core Itaniums.
3. **NUMA machine:** A massive multicore system with a total of 48 cores.

To evaluate the algorithms, they have considered three different graphs:

- US patent citations, with 3.8 million nodes and 16.5 million edges.
- Orkut social network, with 3.1 million nodes and 234.4 million edges.
- A portion of the World Wide Web, with 105 million nodes and 2.5 billion edges.

[1] Triangular loops are nested loops in which the number of iterations of the inner loop depends on the iteration number of the outer loop

The parallel triad census algorithm was implemented using a compact data structure compatible with any of the shared-memory platforms. Graph nodes are stored in a simple array, whose memory is allocated only at the beginning. The edges are also allocated in an array. Each element of the node array contains the node id, the node degree and a pointer to the beginning of the edge subarray where its edges are populated. The edges contain the node tag of the corresponding neighbor, and the two lower bits of the element are reserved for storing the edge direction: '01' indicates that the edge is directed to the neighbor, '10' means the edge comes in to the node, and '11' expresses that the edge is bidirectional. This compact data structure enables some kind of spatial locality. The fact that each node u tracks the edge count allows the compiler to parallelize the control loops that transverse the neighborhood of u. On the other hand, edge subarrays are sorted to enable fast edge searching using binary search.

Another optimization introduced is focused on the manner in which the nodes of a triad are identified and processed. In previous developments, the algorithm created the set S with the union of the neighborhoods of two first nodes, u and v. In this case, instead of creating this set, the authors propose to maintain two pointers that stride through the sorted neighborhoods of u and v, incrementing and setting the value of w to construct the triad (u, v, w). When the two pointers point to a common neighbor, both pointers are incremented after setting w.

Evaluation and results

After setting these optimizations, the authors executed a preliminary test using 8 processors of the Cray XMT on the Orkut network. The CPU utilization was consistently around 60-70%, which is significant taking into account that conventional XMT applications show an average of 30% CPU utilization and do not exceed 50%.

Afterwards, the evaluation was performed across the different shared-memory platforms mentioned above. In order to do so, the XMT code was ported to OpenMP. Both the HP Superdome and the NUMA machine were not able to collapse the nested loops over the graph's vertices and edges in an efficient way, so it was necessary to perform manual modifications of the loops in order to improve balanced workload. Besides, the scheduling policy selected to execute the tests was the one exhibiting the best performance, which turned out to be the *dynamic* scheduling policy.

The first tests were made upon the paper citations' graph. The purpose of the tests was to evaluate the performance using different OpenMP thread counts. For a small number of OpenMP threads (up to 36), the NUMA architecture performs best, due to its memory bandwidth and low latency characteristics. Over 36 threads, NUMA suffers performance degradation, even before the 48 physical core limit is reached. HP Superdome outperforms XMT for a thread count less than 8 (the cell size), but after exceeding this limit, it is the architecture that shows the poorest performance. The Cray XMT machine exhibits the best performance for large thread counts, and also the highest speedups.

The tests over the Orkut Social Network graph show a significant performance improvement for both the HP Superdome and NUMA systems. This fact is attributed to the possibility of masking the unbalanced inner loop workload in this case. However, the results are mainly similar to those of the first graph, since the XMT machine reaches a better performance and speedup for large thread counts.

Finally, the last example of the Web graph could only be executed on the Cray XMT machine, since neither HP Superdome nor NUMA systems could handle such a large graph. Results showed that the algorithm achieved a good linear speedup rates from 64 to 512 processors.

As a conclusion, the authors can extract the following recurrent pattern: Although underperforming with low processor counts, the XMT machine leverages its fine-grained parallelism

capabilities to show great results as the processor count increases. On the contrary, both NUMA and HP Superdome massive concurrency opportunities perform well at small thread counts, but as the thread count grows, their memory systems get overtaxed, thus yielding a poor performance.

Brief Analysis

The most interesting ideas proposed in this work are the management of the data structures used in the triad census algorithm. First of all, the compact way of saving the adjacencies allows to reduce memory usage and search time. Besides, the idea of creating two independent arrays is a simple but effective way to save the graph information avoiding the excessive usage of pointers and references throughout the memory map. Later in this report[2] will be clear that this design has played an important role in the project implementation.

Finally, the way in which the adjacency list union (S) is performed, in-place and exploiting data ordering, allows saving memory and execution time of a basic operation in the algorithm.

2.3.3 Fast Parallel Graph Triad Census and Triangle Counting on Shared-Memory Platforms

In this paper [13], the authors study two graph analysis problems: the triangle counting in undirected graphs[3], and the triad census in directed graphs. Their main contribution is the idea of pre-ordering the vertices of the graph in a convenient way to reduce the operation counts.

- Triangle counting:
  There are many variants of the triangle counting algorithm, which can be classified in two blocks: Adjacency Intersection (AI)-based algorithms and Adjacency Marking (AM)-based algorithms. The first approach intersects the adjacency lists of each pair of nodes (u, v) in G. The second approach marks the adjacent nodes of u, and then checks if some $w \in Adj(v)$ is marked, to count the triangle. It performs a little faster at the expense of consuming a little more memory.
  The paper proposes to construct new enhancements of the algorithm from these two approaches, based on sorting techniques. They define the following sets:

  $$Adj^{+}(u) = \{v \in Adj(u) : v > u\}$$

  $$Adj^{-}(u) = \{v \in Adj(u) : v < u\}$$

  The idea is to compute the intersection of the adjacencies by striding through the sorted adjacency lists and identifying common ones. To avoid counting the same triple multiple times, a canonical ordering $u < v < w$ must be selected, and that is the reason it is only necessary to intersect the sets defined above. The different variants of the AI and AM based methods depend on the choice of what adjacency list to intersect ($Adj^{+} \cap Adj^{+}$; $Adj^{+} \cap Adj^{-}$; $Adj^{-} \cap Adj^{-}$). Theoretically, the authors show that reordering the vertices such that lower degree vertices are assigned low vertex identifiers, and choosing to intersect the sets Adj^{+}, reduces the operation counts.

- Triad Census:
  The authors take the compact data structures and optimizations made by Chin et al. and discussed in section 2.3.2 (which they refer to as the *baseline*), and introduce the vertex reordering mentioned above (both the AI and the AM variants).

[2] Cf. Chapter 4

[3] This problem consists in counting all 3-node complete subgraphs

Evaluation and results

The authors test their proposed improvements across multiple large-scale graphs. Since the reordering overhead turns to be significant with respect to the main computation time (triangle counting/triad census), it is not taken into account for performing the comparison. The different tests were made on two different machines: the dual socket, 8-core Intel Xeon (SNB) and the Xeon Phi (KNC).

- Triad census performance:
  The authors experimented with static and dynamic scheduling with different chunk sizes and came to the conclusion that the chunk-size 10 setting and the dynamic scheduling performed best for the majority of the cases. Results showed that the AI and AM variants provide a significant improvement with respect to the baseline. While the AM variant performs better in the SNB in the majority of the cases, the AI variant performs better in the KNC.
- Triangle counting performance:
  The best scheduling policies were dynamic scheduling with a chunk size of 50 on SNB and dynamic scheduling with a chunk size of 10 on KNC. The comparison between the new variants introduced in the paper and the previous implementations of the algorithm showed that the former provide significant enhancements with respect to the latter. Again, the AM variant performs best in the SNB machine, while the AI is faster than AM in KNC.

In terms of performance scaling, results show that in the triangle counting problem, the AI variant offers the best scaling in both platforms (SNB and KNC). Instead, for the triad census problem, the scalability is comparable for all variants.

Finally, the authors evaluate the impact of ordering on overall performance. In general, results show that the performance of triad census does not improve so much after ordering. Instead, in the case of triangle counting, the introduction of the ordering technique significantly enhances performance.

Brief Analysis

This paper makes the original suggestion of considering order to try to boost algorithm performance. In fact, as read on line 5 of pseudocode 2, the algorithm strides through the set $\{Adj^+(u) : u \in V\}$. Therefore, ordering the nodes in increasing order of number of neighbors could possibly help to reduce the number of iterations of the inner loop. In the case the outer loop is parallelized, the workload among the different threads of execution will be better balanced.

3

OpenCL Overview

This chapter is devoted to the OpenCL framework, which has been used the technology used throughout the project to develop the final application. The explanation focuses on the concepts and tools used, which may also be useful to understand other parts of this report. It also includes some of the programming restrictions which have determined project design decisions, as well as the prototypes of the API functions used. For a more in-depth explanation of the OpenCL specification, see [9] [14] [15].

3.1 What is OpenCL?

As already mentioned in chapter 1, modern high-performance computer systems are not based only on powerful CPUs, but combine different hardware devices (CPUs, GPUs, and FPGAs, among others) that collaborate together to accelerate each part of the application they are executing.

Traditionally, developers in this kind of architectures would write custom code for each device they were working on. For FPGAs, instead, to get the best performance, they would have to design the circuits using an HDL language. This kind of approach, though, is very inefficient in terms of development time. As this paradigm has propagated, it has become clear the need for a standard model for cross-platform parallel computing.

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming on heterogeneous systems. It allows the use of a C-based programming language for developing code across different platform such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), and Field-Programmable Gate Arrays (FPGAs).

OpenCL is based on standard ANSI C (C99) with extensions to extract parallelism. It also includes an Application Programming Interface (API) for the different components to communicate with each other.

3.2 OpenCL architecture

This section presents the architecture and basic concepts of OpenCL. The OpenCL architecture is based on a hierarchy of models.

3.2.1 Platform Model

The Platform Model describes how the physical devices interact with each other. The model consists of a **host** connected to one or more **OpenCL devices**. An OpenCL device is divided into one or more **compute units** which are further divided into one or more **processing elements**. Computations on a device occur within the processing elements.

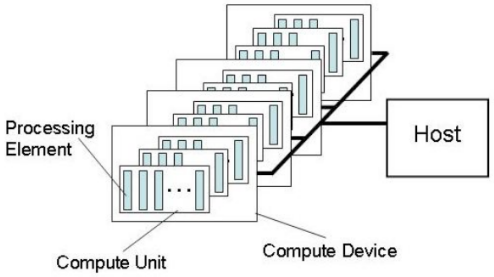

Figure 3.1: OpenCL Platform Model

3.2.2 Execution Model

The Execution Model defines how the execution flows. Execution of an OpenCL application is carried out by two main actors: **kernels**, which execute on one or more OpenCL devices; and a **host program** that executes on the host. Each kernel is associated to a so-called **index space**, which represents the problem that the kernel has to solve (e.g, if the kernel performs the dot product of a vector, the index space world represent the vector itself). When a kernel is launched, the index space is defined. An instance of a kernel is called a **work-item**, and it executes in a point in the index space. Work-items are organized into work-groups. Each work-item has a global id and a local id. Using this execution model, a wide variety of programming models can be implemented. OpenCL supports two: the Data Parallel Programming Model and the Task Parallel Programming Model.

Before launching kernels, the host needs to define the context of execution for them. The context includes four main resources: The **devices** are the collection of OpenCL devices that interact with the host. The **kernels** are the OpenCL functions that run on the OpenCL devices. The **program objects** are the executables that implement those kernels. Finally, the **memory objects** are portions of memory that are visible to the host and to the devices, and on which kernels operate.

3.2.3 Memory Model

The Memory Model considers four possible memory regions: The **global memory** permits read/write access to all work-items in all work-groups. The **constant memory** remains unmodified during the execution of a kernel and allows only read-access on the device side. The **local memory** is a region that allocates variables that can be shared by all work items in a

work-group. Finally, each work item has its own **private memory** region where it stores the variables that are not visible to other work-items.

OpenCL uses a **relaxed** consistency memory model, i.e the state of the memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. Memory consistency can be guaranteed by synchronization mechanisms.

3.2.4 Programming Model

As mentioned before, OpenCL supports two programming models: The Data Parallel Programming Model and the Task Parallel Programming Model.

Data Parallel Programming Model

In a data parallel programming model, the same stream of instructions is applied multiple data elements. The OpenCL execution model defines an index space and a mapping of each point of the index space to a work-item. This mapping, though, is not necessarily one-to-one.

OpenCL provides a hierarchical data parallel programming model, in which data is firstly divided among work-groups and then is furtherly divided among work-items. The programmer can explicitly define the total number of work items and also how the work-items are divided among work-groups, or simply specify the total number of work-items and let the OpenCL framework manage the division among work-groups.

Task Parallel Programming Model

In the OpenCL task parallel programming model, each work-item is executed independently of any index space, i.e, there is not a mapping to a specific portion of the data. Under this model, the programmer can express parallelism by:

- Using vector data types implemented by the device,
- enqueuing multiple tasks, and/or
- enqueing native kernels developed using a programming model orthogonal to OpenCL.

Synchronization

There are two domains of synchronization in OpenCL:

- Work-items in a single work group.
- Commands enqueued to command-queues in a single context.

Synchronization between work items in a work group is done by means of a work-group barrier. When a work item encounters a barrier, it waits until all work items of its work group reach the barrier to resume execution. This can be used, for example, to guarantee memory consistency on that barrier.

There is no mechanism to enforce synchronization among work-items belonging to different work-groups.

The synchronization points between commands in command queues are:

- Command queue barrier: It ensures that, after the barrier, all previously queued commands have finished execution and memory updates are visible to the following commands. This barrier can be used to synchronize between commands in a single command-queue.

- Waiting on an event: All OpenCL API functions that enqueue commands return an event. A subsequent command waiting on that event is guaranteed that updates to the memory objects updated by the command that returns the event are visible before the command begins execution.

3.3 The OpenCL C Programming Language

This section presents the most important the OpenCL C programming language characteristics and restrictions. The OpenCL C programming language is based on the C99 specification, with some extensions and restrictions.

3.3.1 Data types

All conventional scalar data types are supported (`char`, `short`, `int`, `long`, `float`, etc). The OpenCL API defines these types adding the prefix `cl_` to allow usage on the OpenCL application. It is recommended to use the API types on the host code, in order to ensure maximum compatibility.

OpenCL defines built-in vector data types for the types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `float`. A vector data type is defined with the type name followed by a literal value n that defines the number of elements in the vector. Supported values of n are 2, 4, 8 and 16. Examples: `short4`, `ulong16`, `char8`.

3.3.2 Address space qualifiers

OpenCL implements four disjoint address spaces `__global`,`__local`, `__constant` and `__private` (with correspondence to the memory regions defined in section 3.2.3). The address space qualifier may be used in variable declarations to specify the region of memory that is used to allocate the object. Arguments to a kernel function in which an address space qualifier is not specified are stored in the private memory region. Arguments declared as pointers can point to one of the following address spaces only: `__global`, `__local` or `__constant`. Casting or assigning a pointer to address space A to another pointer to address space B is illegal.

3.3.3 Function qualifiers

The `__kernel` qualifier declares a function to be a kernel that can be executed by an application on an OpenCL device. This kind of function can be executed only on the OpenCL device. It can be called by the host program and by other kernel function.

3.3.4 Restrictions

- Arguments to kernel functions that are pointers must be declared with the `__global`,`__local` or `__constant` qualifier

- A pointer declared with the `__constant`, `__local` or `__global` qualifier can only be assigned to a pointer declared with the `__constant`, `__local` or`__global` qualifier respectively.
- Variable length arrays and structures with flexible (or unsized) arrays are not supported.
- Many C99 standard headers (including `stdio.h`, `stdlib.h` and `string.h`) are not available and cannot be included by a program.
- Recursion is not supported

3.4 Basic OpenCL API functions

This section lists the OpenCL API functions used in the project code. For more information regarding input parameters and return values, please refer to the documentation ([9]).

- `clGetPlatformIDs()`: Returns the IDS of the platforms available.
- `clGetDeviceIDs()`: Returns the IDs of the OpenCL devices available.
- `clCreateContext()`: Creates the proper context to execute the kernels.
- `clCreateCommandQueue()`: Creates the command queue for interactions between host and device.
- `clCreateProgramWithBinary()`: Creates the program from the device binary file. In the case of FPGA-based computing, this is the function that loads the HDL file onto the FPGA.
- `clCreateKernel()`: Creates a kernel variable from the program object.
- `clCreateBuffer()`: Creates a buffer object to load/store memory on the OpenCL device.
- `clEnqueueReadBuffer()`: Copies memory from the device to the host.
- `clEnqueueWriteBuffer()`: Transfers memory from the host to the device.
- `clSetKernelArg()`: Specifies a buffer object or variable as an argument for a specific kernel.
- `clEnqueueTask()`: Launches a single work-item kernel on the device.
- `clEnqueueNDRangeKernel()`: Launches an NDRange kernel on the device.

4

System Design and Implementation

This chapter describes the implementation of the Triad Census Algorithm over a heterogeneous parallel environment that exploits FPGA acceleration.

The project development has followed an incremental life cycle: the first steps were focused on obtaining a basic but correct sequential implementation of the algorithm, a necessary baseline upon which to build more sophisticated versions of the algorithm. Continuous performance testing, along with the guidelines offered by the Intel® documentation [14] [15] [16] [17] helped to fine-tune the final version of the application.

The chapter summarizes the process followed, discussing the most relevant design decisions and showing the different variants of the algorithm that have been developed. The explanation follows a chronological pattern, deepening into the design problems encountered along the way, and detailing the ideas and approaches that have led to their solution.

4.1 Application design

4.1.1 Basic functionality required and application flow

The program that implements the algorithms seen in chapter 2 must complete the following general steps:

1. Reads a graph from a file;
2. Performs the triad census on that graph via a user-specified algorithm, and
3. Presents results and execution times.

The file contains the graph in the following format: each line corresponds to one edge, which is represented as two numbers: the source node id and the destination node id.

Note: All graphs considered are unweighted and directed[1]. Self-loops and multi-edges are not allowed.

4.1.2 Data structure design

This section describes the data structures used to store, read and manage the data used within the application.

In the application design, the purpose was to preserve the maximum cohesion within modules and the lowest coupling among them, so that modifications on one module did not affect the others. Therefore, each data structure developed includes interfaces that allow hiding information about implementation details.

In order to select the most appropriate data structures to use, it is important to know what are the data elements that the algorithm needs to compute the triad census. As illustrated in the pseudocode of the algorithms[2], the information required includes the graph nodes, the adjacency lists of each of them, and the direction of each edge. This information is sufficient to calculate the triad class of each triple of nodes within the graph.

Therefore, three data structures were implemented: `graph`, `node`, and `edge`.

The `graph` data structure

The data structure used to save the graph in memory was the **array**. As explained in section 2.3.2, the array is a simple data structure that is allocated contiguous in memory, so it allows exploiting some kind of spatial locality.

In order to give flexibility to the design, dynamic memory allocation has been performed. This technique does not require to know the size of the data in advance and uses the exact amount of memory necessary.

Therefore, as the program reads the graph file, it allocates the space for each of the nodes and saves them, controlling first that the node is not already present (to avoid duplicates). At the end, it stores the number of nodes in an ad-hoc field, since it is a datum that is used recurrently.

The `node` data structure

For each node in the graph, it is necessary to save the information about its adjacency list and its degree. The *Handshaking Lemma*[3] implies that saving the complete adjacency lists of all the nodes means saving the edge information twice. Anyway, this design allows easy and fast access to each edge from both nodes.

The `node` data structure contains four fields. First, the node id, which is implemented as an unsigned integer. Second, the adjacency list of the node. The nodes of the adjacency list are stored in an array of type `edge` which is allocated dynamically. The node structure just saves a pointer to that array. The third field contains the number of the node, also as an unsigned integer. Finally, the four field contains the degree, which can be higher the number of neighbors. This final field is used to calculate the total number of edges in the graph.

[1] the undirected case turns out to be another problem called *triangle counting*, whose study is not within the goals of this work

[2] Cf. section 2.2

[3] Cf. section 2.1

The `edge` data structure

As explained earlier, the `node` data structure needs to save information about its adjacency list. In particular, for each neighbor v of a certain node u, it is necessary to store the neighbor id and the direction of the edge. Following the idea given by the paper analyzed in section 2.3.2, the `edge` data structure is implemented using an unsigned integer: the majority of the bits store the neighbor id, while the two less significant bits store the direction, in the following way:

- '10' indicates that the edge goes for u to v
- '01' indicates that the edge goes for v to u
- '11' indicates that the edge is bidirectional

4.2 Sequential implementation

The initial version of the application was the sequential implementation of the algorithm in C. The goal was to test the adequacy of the data structures previously described, as well as verify the correctness of the different functions that perform the computations of the triad census. This first sequential version would be the baseline for further improvements and optimizations, including parallelization.

The programs were written following the pseudocode presented in section 2.2. Both the BF and the BM algorithms were implemented. Both versions make use of the `isoTricode` procedure, which assigns to each triple of nodes its corresponding class, represented by a number from 0 to 15. In this function, each possible edge within the triple of nodes is represented by a bit (1 if it exists, 0 if it does not). Therefore, the triad configuration is represented as a 6-bit number. The corresponding class code is computed and returned by means of a `switch-case` statement.

Ordering

The sequential implementation includes the possibility to keep data ordered, both the array of nodes and the adjacency list arrays, ascending order of the node id. Ordering occurs at insertion time and is performed using the inner loop of the insertion sort algorithm, which has linear complexity. Maintaining partial results ordered permits to use binary search at any time to check if a new node has already been inserted.

4.3 Heterogeneous implementation

4.3.1 Initial OpenCL version

Once the implementation of the sequential version executed correctly, the initial heterogeneous version was developed. The framework used was OpenCL, with the implementation provided in the *Intel® FPGA SDK for OpenCL* [16]. This SDK provides the OpenCL APIs along with an offline compiler to perform the High-Level Synthesis (HLS) of the code to generate a hardware configuration file. As explained in section 3.2.2, the OpenCL framework allows declaring special functions called *kernels* that execute on the accelerator and upon which perform parallelization and profiling in order to boost performance. In this case, the more complex task in terms of execution time and resource usage is the calculation of the triad census. Therefore, the

first approach was to move the code of the triad census algorithm and wrap it inside a kernel. Following the philosophy of the OpenCL programming model, the idea was to create a host program that accomplished the following steps.

1. Sequentially read the graph and construct the respective graph data structure.
2. Write these data into a buffer and send it to the device
3. Create the kernel from precompiled binary and send it to execution
4. Read the triad census results back from the device.
5. Show the results and/or the execution times

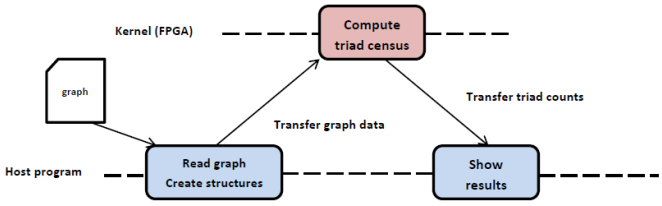

Figure 4.1: Heterogeneous execution flow

These tasks cannot be completed only with the structures and data structures designed for the sequential version due to the fact that OpenCL programming language imposes some restrictions. The usage of pointers is not so flexible as in pure C, and in particular, a kernel argument cannot contain a pointer to pointers. The reason is that OpenCL must convert host addresses (from the host memory) to device addresses. This process is not feasible if the framework does not know in advance how many addresses must translate. Therefore, the compiler does not allow the usage of pointers to structures containing pointers.

As a result, the `graph` structure could not be passed to the kernel code as initially designed. There was the need for new data structures to store the graph information an avoided the intensive usage of pointers. The simplest solution, partly taken from section 2.3.2, was to create two independent arrays: one for the nodes, and the other for the edge. The node structure, instead of having a pointer to the adjacency subarray, keeps track only of the initial and final indices of its adjacency list within the edge array. Figure 4.2 illustrates the new data structures.

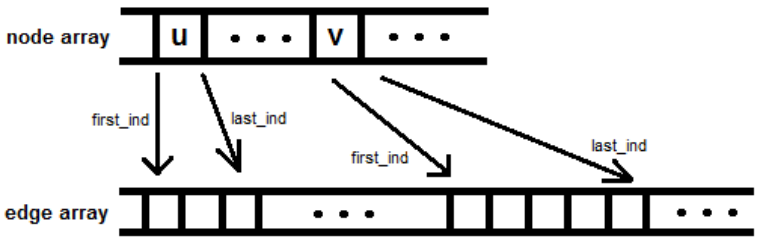

Figure 4.2: Scheme of the new data structures introduced

This implementation allowed to send to the device just two pointers, one to the node list and other to the edge list. These data structures preserve all the graph information needed

to perform the triad census and, at the same time, they are more manageable by the OpenCL framework.

Again, the initial aim was to verify the correctness of the output of the algorithm. This was accomplished by means of the Emulator environment provided by the *Intel® FPGA SDK for OpenCL*. This tool allows emulating a heterogeneous architecture without actually performing the whole synthesis to a hardware configuration file. Thanks to it, validation could be performed in a fast and easy way, without having necessarily a real FPGA attached to the underlying hardware.

4.3.2 Kernel design alternatives

After reaching an initial working version, different variants of the algorithm were explored. The idea was to take advantage of the OpenCL framework utilities and the FPGA design potential to test several implementations and be able to compare the different versions. Thus, three orthogonal criteria were established:

1. The first criterion regards the algorithm itself. Both BF and BM versions were developed. Presumably, the subquadratic version will be much more efficient, due to the complexity of the algorithm, but the simplicity of the BF approach may allow reaching a higher speed-up.

2. The second criterion regards the model of execution of the kernel. As explained in section 3.2.4, the OpenCL framework defines two models of execution of the kernels. The **single work-item kernel** and the **NDRange kernel**. In the case of running an NDRange kernel, the host program launches multiple work-items that execute the same code on different portions of data. On the other hand, single work-item kernels are executed by only one thread. The former exploits *data parallelism*, and the latter exploits *task parallelism*. In this problem, data parallelism can be easily exploited: it is sufficient to divide the work into tasks as explained in 2.3.1, create different partial *census* arrays and then perform a reduction to obtain the final result. Nevertheless, this introduces the overhead of having to create the tasks array before launching the kernel and then merging the results at the end. On the other hand, FPGA architecture is made to boost performance through pipelining techniques. Therefore, it is recommended to design the kernel as a single work-item kernel.

3. Finally, it seemed interesting to explore the differences between the search algorithms applied. There are two possible approaches: linear search and binary search[4]. Obviously, when arrays are ordered (as in this case) binary search performs in logarithmic time, which beats the linear performance of the linear search. Nevertheless, the simple structure of the linear search loop may play an important role in this case. The OpenCL framework allows exploiting massive pipelining when loops do not have data dependencies, reducing the initiation interval (II) to one. This basically means that one iteration of the loop can be launched at every clock cycle, thus increasing the throughput.

Using these three criteria, $2^3 = 8$ different kernels were developed:

- **single_BF:** The Brute Force algorithm implemented as a single work-item kernel, and using linear search to look for data in the arrays
- **single_BF_ord:** Ordered version of the previous kernel (uses binary search instead of linear search).

[4]It was necessary to eliminate recursion from binary search, as it is not supported by the OpenCL standard (Cf. section 3.3)

- **single_BM:** The Bataglej and Mrvar's algorithm implemented as a single work-item kernel using linear search to look for data
- **single_BM_ord:** Ordered version of the previous kernel (uses binary search instead of linear search).
- **NDRange_BF:** The Brute Force algorithm implemented as an NDRange kernel and using linear search to look for data. Each work item performs one iteration of the outer loop (i.e the first node of each triad that computes is always fixed)
- **NDRange_BF_ord:** Ordered version of the previous kernel (uses binary search instead of linear search).
- **NDRange_BM:** The Bataglej and Mrvar's algorithm implemented as an NDRange kernel and using linear search. In this case, the first two nodes of each triad computed by a work item are fixed. Therefore, there are as many work items as the number of pairs $P = \{(u, v) \in E : u \in V, v > u\} \cup \{(v, u) \in E : u \in V, v > u\}$
- **NDRange_BM_ord:** Ordered version of the previous kernel (uses binary search instead of linear search).

A note on synchronization

When working in a parallel environment, it is always possible that memory consistency is affected by different phenomena such as race conditions and other concurrency issues. Therefore, from the programming point of view, it is very important to analyze the possible problems that may arise in problems of this nature and use different mechanisms to prevent them.

In this particular problem, there are two data blocks which could cause problems, as multiple compute units may access them concurrently:

1. The first one is the graph data, which is stored in the form of a node array and an edge array. These data have to be accessed only for reading (it does not have to be modified) and thus can be declared in the `constant` memory area. Accessing this memory concurrently will not produce any consistency problems.
2. The second one is the triad census array in which the triad counts are written. In this case, this array is read-write, so consistency problems will arise if the problem is not handled correctly. To tackle this problem, two policies may be adopted:
   (a) Using only one global array to which all compute units can access, but implementing locks and using atomic primitives to ensure a correct final result.
   (b) Each compute unit writes the results in an independent census array, and then the partial arrays are reduced by sum to obtain the final result.

   The first policy is simple to manage from the host side since the host program needs only to send a single array initialized to 0 and, when the kernel finishes, it receives the final result. On the device side, instead, it is very complex, as multiple kernel instances may have to access the same data element (the triad census array) at the same time. It would be necessary to ensure data integrity, which is a complex problem to cope with, and it decreases significantly the kernel performance because the kernel instances must wait until the resource is free to resume execution. Besides, the software version used for developing the project does not support atomic operations. As a result, the second option was selected. As already mentioned, though, this introduces the overhead of having to

sum all partial results to obtain the total triad census. One of the aims of chapter 5 is to evaluate whether, even with this overhead, an NDRange kernel is more efficient than a single work item kernel.

4.3.3 Optimizing area usage

When performing the High-Level Synthesis of the kernels described above some problems of FPGA area usage arose. The more complex is the algorithm to synthesize, the greater the FPGA area is needed to implement the circuit. In some cases, the tools that perform the synthesis cannot cope with such a complex design, leading to a resource saturation and finally a process halt.

Therefore, the greatest challenge at this point was to identify the parts of the algorithm that would unnecessarily increase the complexity or the resource consumption. It was necessary to perform some modifications on the algorithm design to reduce the kernel logic while preserving an identical functionality.

This process was done by evaluating the results in terms of area usage computed and displayed by the offline compiler when invoking the command with the options `aoc -c -report <kernel_file>.cl`. By using this command, it was possible to evaluate whether each modification in the code would reduce resource usage or not. The approach, therefore, was trial-and-error, but it was also guided by intuition and some tips found in the Intel® documentation.

The first thing worth mentioning is that a significant part of the FPGA area must be used for tasks which have nothing to do with the circuit that implements the algorithm. These logic blocks implement the interfaces necessary to allow the actual circuit to communicate with external devices, and other important functionality required to deploy effectively the circuit on the FPGA. Table 4.1 shows the area used by this logic.

Resource Type	Estimated usage (%)
Logic Utilization	49
ALUTs	23
Dedicated Logic Registers	27
Memory Blocks	23
DSP Blocks	12

Table 4.1: Estimated resource usage of an empty kernel

As Table 4.1 shows, an important part of the FPGA is by default occupied with other logic and components that do not perform any actual computation. Therefore, a great effort must be done to fit the kernel logic on the free space of the FPGA. When implementing the first versions of the algorithm discussed in section 4.3.2, attention was focused on functionality rather than on area optimization. Table 4.2 shows the initial area usage figures.

Resource Type	single.BF	single.BF.ord	single.BM	single.BM.ord	NDR.BF	NDR.BF.ord	NDR.BM	NDR.BM.ord
Logic Utilization	55	55	59	66	55	57	67	88
ALUTs	25	25	26	28	24	25	28	34
Ded. Log. Reg.	31	31	33	38	31	33	40	54
Memory Blocks	29	31	37	49	27	27	33	35
DSP Blocks	12	12	13	13	12	12	12	12

Table 4.2: Estimated resource usage (%) of the initial eight kernels designed

With these figures, it was possible to perform the whole synthesis only for the BF kernels. None of the BM kernels completed the synthesis. With such high percentages of usage, HLS process would halt unexpectedly at a certain point.

In an initial attempt to reduce the complexity of hardware generation, a general analysis of the kernels was performed. The following decisions were taken:

1. **Integer precision reduction:** In all possible cases, long 64b integers were replaced with standard 32b integers, and even short 16b integers. Initially, the 64b precision was used to ensure that even large graphs with a great number of nodes could be managed by the application. Since the node ids and triad counts had to be managed, it was decided initially to use such precision. In any case, using hardware for 64b variables is potentially much more complex than using 32b variables, because all circuit components (comparators, adders, etc.) must support 64b inputs. Unsigned 32b integers allow to manage graphs up to 4 billion nodes, which was considered acceptable for the tests. The only data elements that conserved the 64b precision were the triad census arrays, to avoid problems with large graphs.

2. **Floating-point operations elimination:** Floating-point operations are expensive in terms of hardware usage. In the initial implementation, the BM kernels need to compute the total number of triads, which involves the following calculation: $\binom{n}{3} = \frac{1}{6}n(n-1)(n-2)$. The purpose of this computation is to find the number of null triads from the counts of all the other triad classes[5]. This calculation can be removed from the kernel code and place in the host code, without introducing very much overhead.

3. **Unnecessary code removal:** A deep analysis of the code was conducted to identify and simplify the logic involved in the kernel code. When possible, variables names were reused to avoid superfluous static allocation. Logic was modified to reduce the number of `for` loops and logic conditions.

After all these changes, area usage had decreased but it was not yet possible to synthesize the BM kernels. It was clear that it was necessary deeper, ad-hoc solutions for each kernel type.

In the case of *single_BM*, the key factor that reduced significantly the area and enabled synthesis completion was the reduction of global memory accesses. As explained in the *Best Practices Guide* [17], it is much more efficient in terms of area to read global data at the beginning of the computation, process these data internally (writing results in local variables) and then write final results in global memory at the end. Therefore, there were created the local arrays for the adjacency list of each node, and for the triad census partial results. In each iteration, the kernel fills the local arrays with the necessary graph data, and then it processes these data to calculate the corresponding triad counts, which are written in local memory. At the end, the local triad census is pushed to global memory.
Table 4.3 shows the comparison of the area usage in the *single BM* kernels.

This approach, though, did not help to reduce area usage in the case of*NDRange* kernels. Another strategy was needed to be able to synthesize them. The solution came by modifying the structure of tasks passed to the kernels. As explained in section 4.3.2, the tasks array contained the ids of a couple of nodes (u, v), and then it was the kernel responsibility to search for the information of these nodes in the *node* array. This simplifies the task queue generation but introduces some overhead in the kernel code. To eliminate this overhead, the task queue structure was modified. Now, each cell of the task queue would contain not only the ids but the nodes (u, v) themselves. Thanks to these, it was no longer necessary to pass the `node` array as

[5]Cf. Algorithm 2 in section 2.2

	single_BM		single_BM_ord	
Resource Type	Before	After	Before	After
Logic Utilization	59	58	66	63
ALUTs	26	26	28	28
Ded. Logic Reg.	33	33	38	36
Memory Blocks	37	31	49	29
DSP Blocks	13	12	13	12

Table 4.3: Area reduction (%) in the kernels *single_BM* and *single_BM_ord*

an argument to the kernel, since it was not necessary to search for information in it.
Table 4.4 shows the comparison of the area usage in the *NDRange BM* kernels.

	NDRange_BM		NDRange_BM_ord	
Resource Type	Before	After	Before	After
Logic Utilization	67	58	88	63
ALUTs	28	25	34	27
Ded. Logic Reg.	40	34	54	37
Memory Blocks	33	31	35	31
DSP Blocks	12	12	12	12

Table 4.4: Area reduction (%) in the kernels *NDRange_BM* and *NDRange_BM_ord*

Thanks to these modifications, the synthesis of all BM kernels could complete successfully.

5 Evaluation and results

This chapter describes the different tests and empirical measurements that have been collected during and after the implementation of the algorithm.

The first section is focused on the verification and validation testing. This was a necessary step to prove that every version of the algorithm would output the correct result in each case. The following sections are devoted to the evaluation of the performance of the different versions of the triad census algorithm.

5.1 Verification and Validation

5.1.1 Verification

The verification that every module of the application was executing correctly, many error checks were added in the code. Special attention has been made to the following instructions:

- Operation System calls: The return of the functions that allocate memory or open files have been checked, to verify that the OS does not deny the request.
- OpenCL API calls: Each OpenCL function returns a code that gives information about the nature of the error that eventually occurs. Every time an API function is called, this error code is checked to ensure the function has executed successfully.
- Data structure function calls: The correctness of the data structure methods is done by means of a STATUS enumeration, which contains two values: ERR and OK. Many functions return this data type, which allows checking anomalies in the execution flow.

Using this kind of error checks along with verbose messages helped to identify bugs in the code.

In those cases where a bug was not clearly identified, the `gdb` debugger has been also used as a tool to rapidly isolate the problems and correct them.

5.1.2 Validation

The next step was to check whether the algorithms implemented would compute and output the correct results. For this reason, an automatic test case generator was written. This test case generator creates all possible 64 triad configurations in different files, along with the expected results. After all these files have been created, another program reads the files and computes the triad census of each case, comparing it to the expected result present on the file. In case there is some error, the program prints the error found.

These tests allowed to validate the first sequential versions of the algorithm, both the Brute-Force approach and the subquadratic Batagelj and Mrvar's algorithm. Afterwards, the correctness of the following parallel versions of the algorithm was tested against the sequential implementations. Using a small graph, it was verified that the parallel version generated the same triad census of the sequential one.

5.2 Sequential performance evaluation

Once verified the correct functionality of the sequential versions, the analysis focused on the overall performance of the sequential versions of the triad census algorithm. The aim was to record some time measurements to be able to compare with the following parallel performance, and thus compute the speedup. Also, it was interesting to explore what impact had the introduction of ordered arrays in the sequential execution.

5.2.1 Test cases

To analyze sequential performance, we used pseudo-random generated graphs of different sizes. The decision of using pseudo-random functions was motivated by the need of being able to replicate the experiments on further occasions and compare results.

The random graph generation function receives the number of nodes (n) and the number of edges (m) as inputs and creates the file graph as output. This file can be directly given to the main program to compute the triad census.

The graphs selected for the testing of the sequential version had the following sizes: 50, 100, 200, 400, 600, 800, and 1000 nodes. The number of edges was calculated by multiplying n for some number k, which could be interpreted as the *average degree* of the graph. This technique allowed to regulate the graph density. The tests were performed using two values of k: $\frac{n}{10}$ which generates a dense graph, and constantly $k = 10$, which produces a sparse graph.

5.2.2 Evaluation

On the tests described below, both reading time and triad census execution time were measured and collected, to check the duration differences among the two tasks. The following tables include both the reading and execution times.

Tables 5.1, 5.2, 5.3 and 5.4 shows the execution times collected for the test cases described above. Both the BF and BM versions were tested, in both the ordered and unordered variants.

As can be clearly appreciated, the ordered versions of the algorithm are much faster than the not ordered versions. This occurs mainly because searching is one of the fundamental tasks performed in the algorithm, and binary search has a logarithmic complexity, which outperforms

		Not ordered		Ordered	
N. nodes	N. edges	Reading time	Exec time	Reading time	Exec time
50	500	0.0009	0.012	0.0008	0.009
100	1000	0.003	0.112	0.001	0.073
200	2000	0.012	1.076	0.004	0.619
400	4000	0.044	8.110	0.009	4.441
600	6000	0.085	26.365	0.018	18.112
800	8000	0.176	80.633	0.025	42.873
1000	10000	0.261	143.280	0.037	80.243

Table 5.1: Reading and execution times (in seconds) for sequential BF algorithms upon a sparse graph ($k = 10$)

		Not ordered		Ordered	
N. nodes	N. edges	Reading time	Exec time	Reading time	Exec time
50	250	0.001	0.011	0.001	0.009
100	1000	0.002	0.035	0.002	0.021
200	2000	0.009	0.093	0.003	0.051
400	4000	0.029	0.239	0.009	0.103
600	6000	0.049	0.457	0.012	0.154
800	8000	0.090	0.754	0.020	0.171
1000	10000	0.127	1.119	0.022	0.213

Table 5.2: Reading and execution times (in seconds) for sequential BM algorithms upon a sparse graph ($k = 10$)

		Not ordered		Ordered	
N. nodes	N. edges	Reading time	Exec time	Reading time	Exec time
50	250	0.0005	0.0077816	0.0004	0.0077652
100	1000	0.003	0.088137	0.001	0.0630464
200	4000	0.022	1.32488	0.009	0.689418
400	16000	0.141	19.4393	0.046	5.6107
600	36000	0.520	93.6948	0.093	19.2272
800	64000	1.199	287.872	0.186	47.809
1000	100000	2.072	684.019	0.337	97.9311

Table 5.3: Reading and execution times (in seconds) for sequential BF algorithms upon a dense graph ($k = \frac{n}{10}$)

		Not ordered		Ordered	
N. nodes	N. edges	Reading time	Exec time	Reading time	Exec time
50	250	0.006	0.0023396	0.0005	0.0015756
100	1000	0.003	0.0276808	0.002	0.0159764
200	4000	0.026	0.360843	0.009	0.13924
400	16000	0.176	5.27452	0.049	1.26933
600	36000	0.659	24.434	0.138	4.32319
800	64000	1.45	76.6874	0.263	10.6299
1000	100000	2.82	184.242	0.470	21.1985

Table 5.4: Reading and execution times (in seconds) for sequential BM algorithms upon a dense graph ($k = \frac{n}{10}$)

the linear complexity of linear search. The intuition mentioned in section 4.1.2 is confirmed here.

Another important observation is that the graph reading time is much less than the triad census computation time, which motivates the decision of parallelizing and accelerating this part of the algorithm, as discussed in section 4.3.1.

The execution times of the BM algorithm are much faster and scalable of the BF approach since the triad counts are computed cleverly. Yet, BM scalability is really affected by the density of the graph, as tables 5.2 and 5.4 show. Let us see it through some graphics.

Figure 5.1 shows the execution time of the ordered Brute Force approach as a function of the number of nodes. The red dotted line represents the empirical measurements, while the green line represents the cubic function that best fits the data. The image demonstrates that the Brute Force algorithm has a computational complexity of $O(n^3)$, as mentioned in section 2.2.

Figure 5.1: Brute-Force execution times in terms of number of nodes

Figures 5.2 and 5.3, instead, plot the execution time of the BM algorithm as a function of the number of edges. The goal in this case is to check whether the theoretical complexity $O(m)$ is observed in the experimental results. As figure 5.3 illustrates, the execution times grow following a polynomial with a degree greater than one. Instead, figure 5.2 fits rather well with the linear theoretical curve. The difference is due to the density of the graphs considered in each case. In fact, the first is a dense graph with average degree $k = \frac{n}{10}$. Instead, the second graph is sparse, with average degree $k = 10$. The figure demonstrates that the subquadratic bound discussed in section 2.2 is valid only for sparse graphs.

5.3 Heterogenerous performance evaluation

This section discusses and results collected on the tests performed upon the accelerated versions of the sequential algorithm discussed in section 4.3.2. The aim is to compare the different approaches taken and discover which one performs best for the algorithm we are dealing with.

Figure 5.2: Sparse graph ($k = 10$)

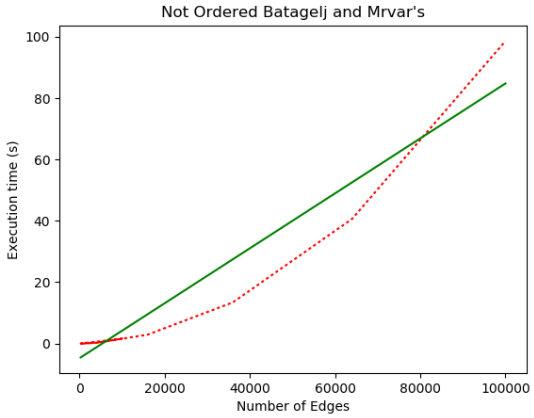

Figure 5.3: Dense graph ($k = \frac{n}{10}$)

Figure 5.4: BM execution times in terms of number of edges

5.3.1 Test cases

In this case, test cases to analyze the BF algorithm are different from those of the BM algorithm. Given the enormous differences in terms of execution times, using small graphs to test performance on BM approaches would yield ridiculously small times, which do not allow to compare and draw reasonable conclusions on execution differences and speedups. On the other hand, using very large graphs increases BF execution times to unacceptable ranges, which slow down the analysis process. As a result, BF analysis has been done using graphs of 50, 100, 200, 300, 400 and 500 nodes; and BM versions have been tested on graphs of 500, 1000, 2000, 3000, 4000 and 5000 nodes.

All tests were made on sparse graphs ($k = 10$).

5.3.2 Evaluation

Tables 5.5 and 5.6 show some results collected from the execution of the BF and BM kernels respectively.

		Not ordered			Ordered		
N. nodes	N. edges	seq BF	single BF	NDR BF	seq BF	single BF	NDR BF
50	500	0.018	0.029	0.006	0.014	0.048	0.005
100	1000	0.116	0.241	0.032	0.093	0.303	0.025
200	2000	0.627	2.926	0.494	0.433	3.356	0.300
300	3000	2.026	10.488	6.259	1.281	12.041	3.490
400	4000	4.731	25.680	16.462	2.848	29.513	9.519
500	5000	8.958	43.948	40.063	5.493	50.915	21.695

Table 5.5: Execution times of the BF kernels

A few conclusions can be drawn from the results shown in the table.

Regarding the ordered vs. not-ordered executions, it is clear that using binary search over ordered arrays outperforms linear search in most of the cases. Only when executing the Brute Force algorithm with the single work-item execution policy, the pipelining of the loops yields better results than using binary search.

		Not ordered			Ordered		
N. nodes	N. edges	seq BM	single BM	NDR BM	seq BM	single BM	NDR BM
500	5000	0.344	0.172	0.105	0.100	0.217237	0.030
1000	10000	1.145	0.383	0.300	0.316	0.453568	0.088
2000	20000	4.202	4.546	0.694	0.695	0.942286	0.207
3000	30000	9.253	14.440	1.101	0.974	1.44186	0.324
4000	40000	16.303	25.260	1.489	1.041	1.94743	0.445
5000	50000	25.246	39.029	1.888	1.495	2.48036	0.562

Table 5.6: Execution times (in seconds) of BM algorithms

On the other hand, these tables show that single work-item executions are not suitable for the kind of application we are dealing with. In fact, in all variants of the algorithm, the sequential executions outperform the single work-item executions. The reason for this can be found in the irregularity that the algorithm presents in various aspects. Memory accesses are not sequential and do not follow regular patterns, preventing the device scheduler from organizing the memory blocks to perform efficient memory usage. On the other hand, algorithm logic is rather complicated and the majority of the loops in the kernel code cannot be pipelined due to exit conditions that are impossible to predict or discover at circuit generation time. As a result, the hardware circuit generated by the HLS is probably rather complex and does not add a significant improvement in terms of throughput.

Instead, the NDRange versions of the BM algorithm present some interesting results. It seems that the NDRange execution model is more suitable to this kind of algorithm. In fact, the BM algorithm performs the same set of instructions over a large quantity of data, which can be divided into different portions that are processed among different kernel instances. This allows exploiting data parallelism.

N. nodes	seq-BM	NDR-BM	speedup
500	0.344	0.105	3.276
1000	1.145	0.300	3.817
2000	4.202	0.694	6.055
3000	9.253	1.101	8.649
4000	16.303	1.489	10.949
5000	25.246	1.888	13.372
10000	125.544	3.935	31.904
15000	280.843	6.072503	46.252
20000	497.564	8.068149	61.671

Table 5.7: Speedup of NDRange kernel execution over not ordered data

Tables 5.7 and 5.8 show the speedups observed in the NDRange kernel executing the BM algorithm. In the not ordered case, speedups are very large and grow with the size of the graph. Instead, in the ordered case, speedup seems to become steady around the value 2.3. This may be due to the Gustafson's Law:

$$S_{lat} = 1 - p + ps$$

The parallelizable portion p (i.e. the algorithm) is much more significant in the not ordered case than in the ordered case. In the not ordered case, as s grows, ps grows significantly and thus S_{lat} becomes very high. When p is very little (ordered case) the ps factor does not determine the value of S_{lat}.

N. nodes	seq-BM-ord	NDR-BM-ord	speedup
500	0.100	0.030	3.333
1000	0.316	0.088	3.590
2000	0.695	0.207	3.357
3000	0.974	0.324	3.006
4000	1.041	0.445	2.339
5000	1.495	0.562	2.660
10000	2.624	1.143	2.296
15000	4.122	1.766	2.334
20000	5.616	2.374	2.366

Table 5.8: Speedup of NDRange kernel execution over ordered data

The best results in terms of execution time have been observed in the *NDRange_BM_ord* kernel. In any case, the way in which portions of data are divided among the different kernel instances is not optimal, since the partition does not take into account the degrees of the nodes that are assigned to each instance. This fact probably affects speedup negatively. As a result, it could be interesting to improve the way of dividing the problem in order to enhance application scalability.

6

Conclusions and Future Work

This project has served as an introduction to FPGA-based heterogeneous computing and to the OpenCL framework as a tool to develop applications in such environments.

FPGAs are powerful devices that can enhance significantly the performances of algorithms and applications, but it is necessary a non-negligible quantity of knowledge and time to exploit all their potential. On one hand, it is important to understand the FPGA architecture and how the basic operations performed by the source code are translated into a digital circuit design to maximize resource exploitation. On the other hand, knowing the nature and characteristics of the algorithm is key to determine how the physical resources can be best organized and distributed among the different tasks.

Fortunately, nowadays there exist tools and methods that facilitate working in such environments. OpenCL is a well-documented, easy-to-use programming model that allows developing heterogeneous applications in CPUs, GPUs, DSPs and/or FPGAs using a common API. Furthermore, the *Intel® FPGA SDK for OpenCL* not only provides an OpenCL implementation, but also a range of tools to debug, emulate, test and profile the accelerated code by using a high-level programming language such as C, instead of having to design the circuit in an HDL.

Regarding the Triad Census Algorithm implemented and, more in general, applications characterized by irregular memory access patterns, the main conclusion that is clear after developing this project is that **pipelining and ad-hoc circuit design are not sufficient to mitigate the problems of memory access latencies**. In fact, these techniques are best exploited when all the information is known at hardware generation time: loop exit conditions, memory accesses, etc. Without being able to predict the behavior of the application in advance (for example, by discovering a certain regular pattern), it is impossible to generate a hardware circuit that actually accelerates frequency and increases throughput. This is the reason why the triad census executed on a single work-item kernel performs so poorly.

Anyway, this is not the only way in which FPGAs are used. Data parallelism can be exploited because hardware replication is also possible. As the results of chapter 5 show, NDRange Kernels outperform the sequential implementations, since they exploit parallelism in the triad census computations.

Nevertheless, the results obtained in this project are very far from the state of the art, mainly because parallelism has not been optimally applied. Fine-tuning an application of these charac-

teristics is a delicate task that requires a lot of time and a great period of testing. Unfortunately, this project did not have so many resources to achieve this.

As a result, a straightforward future development could be to optimize the application developed in this project, by trying different techniques (including those discussed in sections 2.3.1, 2.3.2 and 2.3.3) and test the developments upon real-world networks, to check whether the FPGA-based implementation can compete with the current fastest implementations discussed in chapter 2.

Glossary

- **Brute Force algorithm (BF)**: Cf. section 2.2
- **Batagelj and Mrvar's algorithm (BM)**: Cf. section 2.2
- **Field-Programmable Gate Array (FPGA)**: Sophisticated hardware device whose logic can be programmed by means of a hardware description language. Cf. Appendix A.
- **Graphical Processing Unit (GPU)**: Coprocessor specially dedicated to the processing of graphics and floating-point operations.
- **Digital Signal Processor (DSP)**: Specialized microprocessor whose design has been optimized for of high-speed numerical operations execution.
- **Software Development Kit (SDK)**: Set of software development tools that allow the creation of applications for a certain framework of hardware platform.
- **High-Level Synthesis (HLS)**: The process of translation of an algorithm developed in a high-level software programming language (such as C/C++) to a circuit written on an HDL.
- **Hardware Description Language (HDL)**: Computer language used to design the behavior of electronic circuits, and, in particular, digital logic circuits.
- **Application Programming Interface (API)**: Specification of a set of function prototypes, modules, and interfaces that allow the programmer to interact with a library.
- **Parallelism**: Technique used in computer system architectures by which a problem is divided into multiple, smaller tasks that are executed concurrently. There exist several approaches to parallelism:
  - **Data Parallelism**: Parallelization technique in which the compute units process different portions of data.
  - **Task Parallelism**: Parallelization technique in which the compute units execute different instructions (possibly on the same data).
  - **Instruction Level Parallelism**: Parallelization technique in which single instructions are executed in parallel.
- **Single Instruction Multiple Data (SIMD)**: Data parallelization technique in which multiple compute units perform the same instruction over different portions of data.
- **Pipelining**: Technique used in computer system architectures by which the dataflow of the microprocessor (or, in general, any digital logic circuit) is divided into multiple stages which can process which can process different instructions independently
- **Initiation Interval (II)**: In the context of pipelined computer architectures, the initiation interval refers to the number of clock cycles that elapse from the issue of a certain instruction (or loop iteration) and the following one.

- **Throughput**: The maximum rate at which a program or process reaches to produce results.
- **Speedup**: A measure of the performance difference between two systems, programs of algorithms, say T_1 and T_2 solving the same problem. It is calculated as:

$$speedup = \frac{T_1}{T_2}$$

- **(FPGA) Area**: Refers to the percentage of resources used on the FPGA device.
- **Race condition**: In the context of synchronization and memory consistency, the race condition refers to a situation in which the results of a certain program depend on the order its instructions are executed.

Bibliography

[1] A.L. Barabási and E.Bonabeau. Scale-free networks. *Scientific American*, pages 50 – 59, 2003. Available at: `http://www.http://barabasi.com/f/124.pdf`.

[2] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, Cambridge, UK, 1994.

[3] B. Batagelj and A.Mrvar. Role based access control design using triadic concept analysis. *Journal of Central South University*, pages 3183 – 3191, 2016. Available at: `https://link.springer.com/article/10.1007/s11771-016-3384-6`.

[4] Ravindranath Madhavan Devi R. Gnyawali Jinyu He. Two's company, three's a crowd? triads in cooperative-competitive networks. *Academy of Management Journal*, pages 918 – 927, 2004.

[5] June Cotte and Stacy L. Wood. Families and innovative consumer behavior: A triadic analysis of sibling and parental influence. *Journal of Consumer Research*, pages 78 – 86, 2004.

[6] Robert Vianello and Zvonimir B.Maksić. Triadic analysis of substituent effects: gas-phase acidity of para-substituted phenols. *Tetrahedron*, pages 3402 – 3411, 2006.

[7] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[8] Intel Corporation. *Introduction to Parallel Computing with OpenCL on FPGAs*. 26-minute online course. Available at: `https://www.altera.com/support/training/course/oopncl100.html`.

[9] *OpenCL 1.0 Specification*. Technical report, Khronos Group, June 2009. Available at: `https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf`.

[10] B. Batagelj and A.Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, pages 237 – 243, 2001.

[11] S. Choudhury G. Chin, A. Marquez and K. Moschhoff. Implementing and evaluating multithreaded triad census algorithms on the cray xmt. *Proc. IEEE of the* $23^r d$ *International Parallel & Distributed Symposium (IPDPS)*, 2009.

[12] S. Choudhury G. Chin, A. Marquez and J. Feo. Scalable triadic analysis of large-scale graphs: Multi-core vs multi-processor vs. multi-threaded shared memory architectures. *Proc. International Conference on Computer Architectures & High Performance Computing (SBAC-PAD)*, 2012.

[13] G. M. Slota S. Parimalarangan and K. Madduri. Fast parallel graph triad census and triangle counting on shared-memory platforms. *IEEE International Parallel & Distributed Processing Symposium Workshops*, 2017.

[14] Intel Corporation. *Writing OpenCL Programs for Intel FPGAs*. 54-minute online course. Available at: `https://www.altera.com/support/training/course/oopncl200.html`.

[15] Intel Corporation. *Running OpenCL on Intel FPGAs*. 47-minute online course. Available at: `https://www.altera.com/support/training/course/oopncl300.html`.

[16] *Intel FPGA SDK for OpenCL Programming Guide*. Technical report, Intel Corporation, Nov 2017. Available at: `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf`.

[17] *Intel FPGA SDK for OpenCL Best Practices Guide*. Technical report, Intel Corporation, Nov 2017. Available at: `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf`.

[18] Scott Hauck and André Dehon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kauffman publishers, 2008.

[19] Intel Corporation. Basics of programmable logic: Fpga architecture. 34-minute online course. Available at: `https://www.altera.com/support/training/course/odsw1006.html`.

A

FPGA overview

This appendix introduces the FPGA as an accelerator device, explaining its architecture and the main concepts of the FPGA heterogeneous computing. For more information, see [18] [19].

An **FPGA** (*Field Programmable Gate Array*) is a sophisticated and flexible hardware device which falls into the so-called **programmable logic devices**. These devices are composed of a large amount of basic **building blocks** whose interconnection can be programmed by means of a Hardware Description Language (HDL). They have the fundamental advantage of allowing the programmer to generate custom hardware to accelerate algorithms. They can also be used to integrate complex systems on a single chip, thus reducing cost and power usage.

A.1 FPGA architecture

As its name indicates, an FPGA is an array of multiple, identical building blocks, which are called **Logic Elements**. These blocks are disposed within groups called **Logic Array Blocks** (LABs). Between the LABs, there are multiple **Routing Channels** that connect the different modules allowing to create complex hardware. An FPGA can be connected to external devices thanks to the **I/O Pads**.

A.1.1 Logic Elements

The basic component of an FPGA is the Logic Element. Logic Elements are furtherly composed of three main parts: the LookUp Table (LUT), the carry logic, and the output register logic.

Lookup Tables

The LUT is one of the main components of a Logic Element. Containing 4 input lines, it allows creating combinational functions under the form of a sum of products (minterms). It is made up of a series of cascaded multiplexers where the LUT inputs are used as the select lines. The logic is called a lookup table because the output is calculated by ”looking up” the correct

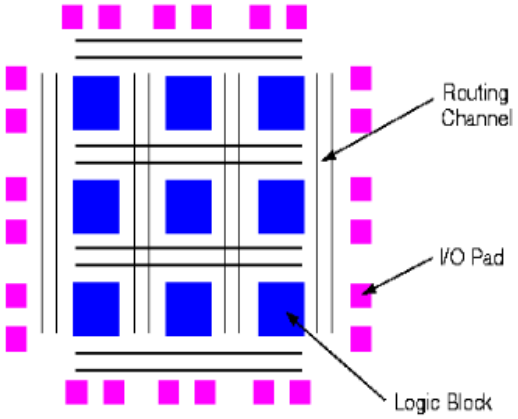

Figure A.1: FPGA structure

programmed levels and routing them through the multiplexers based on the LUT input signals. The programmed levels selected are based on the truth table of the function.

Programmable Register

The programmable register composes the synchronous part of an LE. It is usually driven by a global clock, but in fact, it can be controlled by any clock domain. Asynchronous functions such as clear, reset, or preset, can be generated by another logic component of the LE, or come from an external interconnect via an I/O pin. The output of this register can be fed back the LUT or go out the LE. In case a strictly combinational function is needed, this register can be bypassed. This increases the flexibility of the LE output.

Carry and Register Chains

The final component that distinguishes the LE architecture from the previous ones is the carry and register chain logic. In previous architectures like CPLDs, the output and carry bits of the macrocells could serve as input to other macrocells, but this would require going back to the product term array to pick up the results. FPGA LEs contain specific carry logic and register chaining routes within LABs to provide shortcuts for these signals. Generated carry bits can be output to other LEs or to the device interconnect. This brings about an enhancement in versatility and therefore provides further performance.

The LE contains the necessary circuitry to only make use of the LUT or the Programmable Register. The output from these components can be directly addressed to anywhere else in the device (either the same LAB or through the device routing channels). Therefore, FPGA LEs can be configured to perform a function called **Register Packing**. With this technique, two different functions can be output from a single LE, if the first one only uses the LUT and the second one only uses the register. This can help to optimize resource usage.

A.1.2 Adaptive Logic Modules

LEs provide a great flexibility and power to create many functions. Nevertheless, an isolated LE cannot create very complex functions. Therefore, techniques as cascading and feedback are

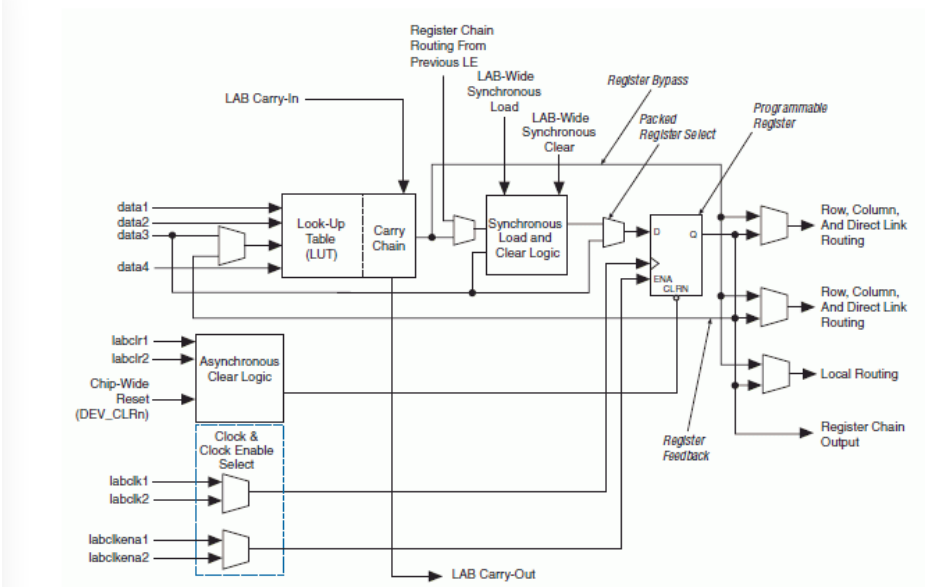

Figure A.2: Structure of a Logic Element

necessary. To overcome this problem, modern FPGAs contain an evolution of the LE called Adaptive Logic Module (ALM). This includes more registers and other dedicated resources to exploit the idea of register packing and to allow generating even more functions within a building block. For example, simple arithmetic functions can be performed in a dedicated adder, avoiding using the LUT.

In particular, the key enhancement introduced in the ALM is the Adaptive LUT (or ALUT). It is similar to a LUT, but it has 8 inputs instead of 4, and these can be split and configured to accommodate two separate functions of any type.

All these enhancements provide extremely high-performance logic operations using few resources and smart resource management.

A.1.3 LABs and Routing Channels

As the previous section explains, the LEs and ALMs are grouped into LABs. These labs are arranged into a large array. FPGA power is mainly due to he Routing Channels that connect the LAB, which can be programmed and reprogrammed yielding different hardware systems.

The Routing Channels provide very much functionality and connectivity. They allow all device resources to communicate with any other resource within the chip, thus making communication very flexible and powerful. The routing is organized as row or column interconnect and includes different-sized fixed length segments, which allows adjusting routing to meet design timing requirements. The number of routing channels scales linearly with the number of resource components needed.

A.1.4 I/O Elements

FPGA I/O control is contained in blocks placed in the external edges of the array, as seen in Figure A.1. They are available for every device component thanks to the routing channels. I/O elements not only contain basic input, output, and bidirectional signaling. They also support a wide variety of I/O features such as multiple I/O standards, differential signaling, etc.

A.1.5 FPGA Advanced Features: Memory and DSP Blocks

Modern FPGAs include special hardware resources apart from the basic blocks explained above. These blocks are constructed with multiple basic blocks and are completely accessible through the routing channels. Explanation will focus on Memory Blocks and DSP Blocks.

Memory blocks can be configured into different types of memory devices: single or dual-port RAMs, programmed ROMs, FIFO buffers, etc. These blocks are programmed just like any other resource in the FPGA, and therefore they can be initialized with any memory content at powerup.

DSP blocks contain embedded multipliers and adders to perform arithmetic and multiply/accumulate operations. They can be used instead of the LE/ALM logic to perform such operations, thus improving arithmetic performance in a certain design.

A.1.6 FPGA clocking structures

Since FPGAs are based on synchronized register logic, clock control structures are important parts of an FPGA architecture. All FPGA devices include dedicated clock input pins, which allow to receive a clock signal from an external device and connect it directly to the clock control structures on the chip. When not used for clocking, clock input pins can be used as standard I/O.

Clock input pins often are input to logic structures called Phase Locked Loops (PLLs). PLLs can be programmed to generate different clock domains based on the input, and ensure that all clock domains are in phase with each other.

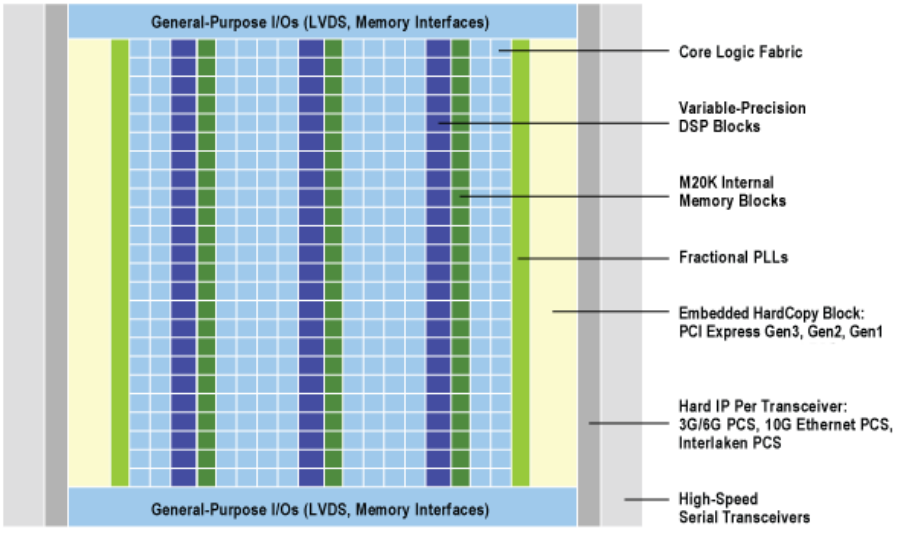

Figure A.3: Intel® Stratix V FPGA Full Chip Architecture

A.2 FPGA Programming

In order to orchestrate all the different architectural components mentioned in the section before, very much programming information is needed. Also, to support possible connections between every component of the logic design, simpler and smaller programming structures are needed. To meet these requirements, FPGAs use SRAM cells to program logic levels and make routing connections.

A row/column interconnect junction within an FPGA includes switching transistors on all possible connections between the vertical and horizontal routing. The select gate control on each transistor comes from an SRAM cell, which is basically a latch containing a programming bit. Depending on the programming bit present in the cell, the corresponding connection is closed or not.

Obviously, this type of programming architecture requires many transistors. However, these transistors are all standard. The main problem of this type of programming is that memory is volatile. Whenever powered off, the SRAM cell is cleared. Therefore, FPGA devices must always be programmed at power-on to configure the device SRAM cells. In order to store the programming information persistently, an external, non-volatile device is used. For example, an EEPROM or a CPU. At power on, the FPGA interacts with this external device to get configured. This step can be accomplished in two ways:

- **Active:** FPGA controls programming sequence automatically at power-on.
- **Passive:** An intelligent host (typically a CPU) controls programming.

A.3 Conclusions

Taking into account all the features described in the previous sections, it is easy to highlight some of the most important advantages that provides the usage of FPGAs.

1. **Multiple logic:** FPGAs contain a lot of user logic to enable the creation of all kinds of logic and arithmetic functions, from the simple to the very complex.
2. **Flexibility:** The great quantity of interconnections present in the FPGA allows to create flexible custom hardware to satisfy any user need.
3. **High Performance at low cost:** In FPGAs, system design is optimized to boost performance. They offer the same features and speeds as other more expensive devices dedicated logic chips (like ASICs).
4. **Many available I/O standards:** FPGA I/O is extremely flexible and can be customized for a specific application.
5. **Fast programming:** The SRAM technology enable the FPGA to be programmed quickly, making the disadvantage of required programming at powerup almost a non-issue.

B

FPGA programming flow

This appendix explains briefly the compilation flow of a heterogeneous application developed using the *Intel® FPGA SDK for OpenCL*. For more information, check the Altera documentation [16].

As explained in chapter 3, OpenCL application has two main components. The host code, which is written in C and executed in a normal CPU, and kernel code, which contains the logic that will be executed on the OpenCL devices (which in this case are the FPGAs).

Therefore, the compilation involves two main steps:

- The host code compilation, which gets compiled with a standard C compiler. The result of this compilation is the host binary.
- The OpenCL kernel code compilation. In this case, this step uses a special tool included in the SDK, called offline compiler, which performs the High-Level Synthesis (HLS). This process is very long and consumes many resources. The process generates two files:
  - `.aoco` file: Object file that contains information for later stages of the compilation.
  - `.aocx` file: Executable file containing the binary used to program the FPGA.

After compilation process has finished and everything is ready, the host binary is executed on the CPU. The host code is in charge of loading the kernel binary file onto the FPGA. The interactions between the host machine and the FPGA is made through a PCIe bus.

Figure B.1 shows graphically the compilation process.

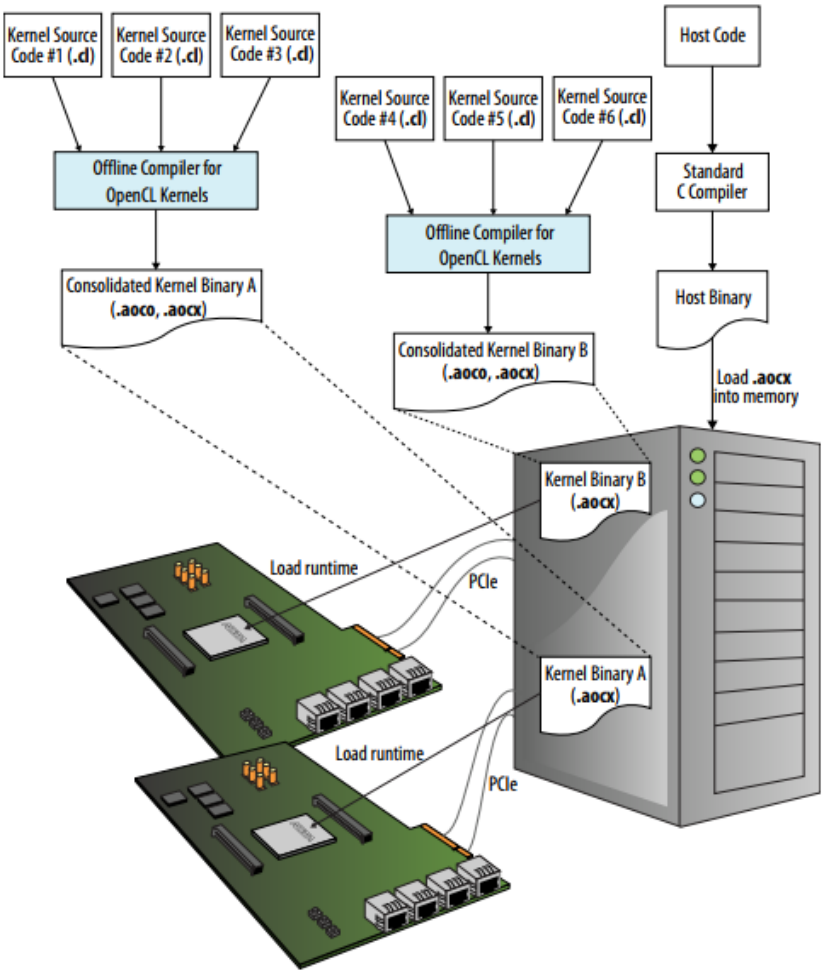

Figure B.1: *Intel® FPGA SDK for OpenCL* Programming Flow

C

User Manual

This appendix explains the necessary steps to download and execute the source code of the project developed. The project has been developed and tested on a Linux system, and therefore it is assumed the project will be deployed in such environment. Correct functionality using other OS is not guaranteed.

C.1 Prerequisites

In order to be able to deploy the project correctly, the user must have installed the following tools.

1. **C compiler included with GCC.**

2. **Intel® FPGA SDK for OpenCL:** The *Intel® FPGA SDK for OpenCL* is the implementation of the OpenCL standard that has been used to develop the heterogeneous implementation of the triad census algorithms. The SDK can be downloaded from the Altera website[1]. The latest version is 17.1, even if this project has used version 16.0 because it was the software version installed on the machine in which the tests have been deployed. The SDK version and the software edition will depend on the target FPGA used in the design.

   In order to download and install the SDK, it is useful to check the installation guide[2] provided by Intel®. It is important to set up correctly all the necessary environment variables before trying to run any code. In order to test software installation, the Altera website provides some design examples[3]. After having tested that those examples execute correctly, the installation process should have ended successfully.

[1] `http://dl.altera.com/opencl/17.1/?edition=pro&download_manager=dlm3`

[2] `https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf`

[3] `https://www.altera.com/products/design-software/embedded-software-developers/opencl/developer-zone.html#design-examples`

C.2 Cloning the repository

The source code is available on a public Github repository. To clone it on your local machine, use:

```
git clone https://github.com/carlosalfaro94/triad_census_on_FPGA.git
```

The project contains various folders:

- **scripts:** Contains several bash scripts used to collect data regarding performance
- **sources**: Contains all C source files (.c files), including main programs. The main programs are:
  - `main_sequential.c`: To run sequential algorithms.
  - `main_parallel.c`: To run accelerated algorithms.
  - `times_sequential.c`: To collect time data from sequential algorithms.
  - `times_parallel.c`: To collect time data from accelerated algorithms.
  - `rand_graph_generation.c`: To generate a pseudo-random graph and save it to a file.
- **headers**: Folder containing header files (`.h` files)
- **exe**: Folder to save executable files
- **kernels**: Contains OpenCL source code (`.cl` files)
- **graphs**: Folder to save the graph files
- **times**: Folder to save the files collecting times
- **doc**: Contains project documentation (generated with `doxygen`)

Finally, the main folder contains the `makefile` to compile all the necessary files.

C.3 Running the sequential algorithms

The C source file called `main_sequential.c` executes the sequential versions of the triad census algorithm. To compile it, run

```
make sequential
```

on the command prompt. The executable file will be saved on the **exe** folder. The sequential versions of the algorithm are two: Brute-Force and Batagelj and Mrvar's (Cf. section 2.2), which can be executed over ordered or not-ordered arrays. The executable must receive as an argument the input graph and the algorithm name to use. The main program also allows executing in verbose mode (showing runtime information) and show only the results or time performances. In order to see execution parameters, run the executable with the `-help` option.

C.3.1 Generating a random graph

To compile the random graph generation program, run `make rand`. Then, run

```
exe/rand_graph_gen <n> <m> graphs/<output_file>.txt
```

to generate the a n-node, m-edge graph and save it in `graphs/<output_file>.txt`.

```
alfaro@sachiel:~/final/triad_census_on_FPGA$ exe/main_sequential -g graphs/prueba.txt -f BM -o  -v
Graph read in 0.000190 seconds.
Graph summary
        - Number of nodes: 7
        - Number of edges: 14
        - Number of triads: 35
Triadic census performed in 0.000046 seconds.
Triadic census:
1 - 003: 2
2 - 012: 10
3 - 102: 2
4 - 021D: 4
5 - 021U: 3
6 - 021C: 3
7 - 111D: 1
8 - 111U: 4
9 - 030T: 3
12 - 120D: 1
13 - 120U: 2
Execution finished successfully
```

Figure C.1: Example of sequential execution

C.4 Running the accelerated algorithms

To run the accelerated algorithms, a more complex process is needed, since they are deployed in an heterogeneous environment that includes an FPGA. To program the FPGA, it is necessary to generate a binary file (`.aocx` file) from the kernel code. This file will be used to program the interconnections of the FPGA.

C.4.1 Running accelerated algorithms on an emulated architecture

Since the High-Level Synthesis is a very complex and long process, the *Intel® FPGA SDK for OpenCL* includes an emulation tool that allows testing the heterogeneous applications without having to generate the binary file. This emulation environment turns very useful for debugging. To generate the emulated binary file from the OpenCL source file, you must run the command

```
aoc -v -march=emulator <source_file.cl> -o <binary_file.aocx> -I headers/ board <board_name>
```

This creates the emulated binary file that will be passed to the main program.

To execute the accelerated application using the emulator, generate the `main_parallel` executable using the command `make parallel`. Then, use the command

```
env CL_CONTEXT_EMULATOR_DEVICE_ALTERA=1 ./main_parallel <params>
```

where the params are very similar to the sequential case, except that instead of specifying the function to use (BM/BF) you have to pass the binary `.aocx` file and the name of the kernel that implements the corresponding version of the triad census algorithm. Again, you can use the `-help` option to obtain parameter information.

C.4.2 Kernel synthesis and FPGA deployment

After having ensured that kernel code runs successfully on the emulation environment, synthesize the kernels to generate the real binary files. For this purpose, use the command

```
aoc -v <source_file.cl> -o <binary_file.aocx> -I headers/ board <board_name>
```

Remember that this process will take various hours to complete. If you want to generate just the object file (`.aoco`) to check performance FPGA usage and optimization reports, you can run the command with the `-c` option. It will take just a few minutes.

To deploy the generated binary file over the FPGA, run the command

```
./main_parallel <params> ...
```

similarly to the emulator case.

C.5 Collecting performance data

The project folder also contains some bash script files that have been used to collect the performance data showed in chapter 5. You can run these scripts to replicate the tests made. They make use of the programs `times_sequential.c` and `times_parallel.c`. Depending on what tests you want to do, you can comment out some lines, modify the sizes of the graphs, or the number of iterations.

You may have to give execution permission to the scripts, using

```
chmod +x <script_name>
```

Their usage is also explained in the comments found in the files.